Ecole Centrale de Nantes Université de Nantes Ecole des Mines de Nantes

ECOLE DOCTORALE STIM

« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATERIAUX »

Année 2010

N° attribué par la bibliothèque

Contribution à l'élaboration d'architectures logicielles à hiérarchies multiples

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE NANTES

Discipline : Informatique

Présentée et soutenue publiquement par

Abdelkrim AMIRAT

Le 29 Septembre 2010 à l'UFR Sciences et Techniques, Université de Nantes,

devant le jury ci-dessous

Président	:	Mme Regine LALEAU, Professeur	Université Paris-Est Créteil
Rapporteurs	:	Mme Laurence DUCHIEN, Professeur	Université de Lille 1
	:	M Jean-François SANTUCCI, Professeur	Université de Corse
Examinateurs	:	Mme Regine LALEAU, Professeur	Université Paris-Est Créteil
	:	M Yvon TRINQUET, Professeur	Université de Nantes
	:	M Mourad OUSSALAH, Professeur	Université de Nantes

Directeur de thèse : Professeur Mourad OUSSALAH
Laboratoire : **L**ABORATOIRE D'**I**NFORMATIQUE DE **N**ANTES **A**TLANTIQUE (LINA)
CNRS UMR 6241. 2, rue de la Houssinière BP 92208 44322, Nantes, France, Cedex 3.

N° ED 503-093

CONTRIBUTION À L'ÉLABORATION D'ARCHITECTURES LOGICIELLES À HIERARCHIES MULTIPLES

Contribution to the development of multiple hierarchies' software architectures

ABDELKRIM AMIRAT

Favet neptunus eunti

Université de Nantes

Abdelkrim AMIRAT

Contribution à l'élaboration d'architectures logicielles à hiérarchies multiples

xvi+172

Remerciements

Au terme de ces quatre années de thèse passées au LINA, c'est avec une immense joie que j'écris ces quelques lignes de remerciements.

Je tiens à exprimer ma gratitude envers mon directeur de thèse le Professeur Mourad Chabane Oussalah pour m'avoir donné la chance de faire une thèse sous sa direction, pour la qualité de ses conseils, sa disponibilité ainsi que le degré de responsabilisation de son encadrement. Il a cru en mes capacités, et il m'a donné les opportunités pour développer et prouver mes compétences pour la recherche.

J'aimerais exprimer ma gratitude à Madame Laurence Duchien, Professeur de l'Université Lille 1 et Monsieur Jean-François Santucci, Professeur de l'Université de Corse, pour avoir consacré un temps précieux à la lecture détaillée de mes travaux, et à Madame Regine Laleau, Professeur de l'Université Paris-Est Créteil et Monsieur Yvon Trinquet, Professeur de l'Université de Nantes pour m'avoir fait l'honneur de participer à mon jury de soutenance en tant qu'examinateurs.

D'autre part, rien n'aurait été possible sans l'effervescence intellectuelle et amicale qui anime le laboratoire LINA. C'est ainsi que tout naturellement, je remercie non seulement l'ensemble des chercheurs seniors, des doctorants, mais aussi l'ensemble du personnels du laboratoire LINA qui ont apporté, directement ou indirectement, leur part de contribution à ce manuscrit, par leurs compétences, leurs disponibilités et leurs gentillesses.

Sur un plan plus personnel, je tiens à remercier ma femme, pour son soutien et son aide.

Table des matières

Table des figures

Liste des tableaux

Introduction

Cadre de la thèse

Cette thèse s'inscrit dans le domaine des architectures logicielles et plus précisément dans celui des langages de description des architectures à base de composants. L'objectif de l'architecture logicielle est d'offrir une vue d'ensemble et un fort niveau d'abstraction afin d'être en mesure d'appréhender la complexité d'un système logiciel [Medvidovic et Taylor 2000]. L'architecture propose une organisation grossière du système comme un assemblage de composants logiciels [Perry et Wolf 1992]. Clairement, la discipline de l'architecture logicielle a favorisé la modélisation de systèmes logiciels de plus en plus complexes.

Actuellement, un grand intérêt est porté au domaine de l'architecture logicielle [Taylor *et al.* 2009]. Cet intérêt est motivé principalement par la réduction des coûts et des délais de développement des systèmes logiciels. En effet, on fournit infiniment moins d'effort à acheter (et donc à réutiliser) un composant au lieu de le concevoir, le coder, le tester, le déboguer et le documenter. Une architecture logicielle modélise un système logiciel en termes de composants et d'interactions entre ces composants, elle joue le rôle de passerelle entre l'étape d'ingénierie des besoins du logiciel et l'étape de conception détaillée du logiciel. Enfin, l'architecture logicielle permet d'exposer, de manière compréhensible et modulaire, la complexité d'un système logiciel.

Pour décrire l'architecture d'un logiciel et modéliser ainsi les interactions entre les composants d'un système logiciel, deux principales approches ont vu le jour depuis quelques années : la *modélisation par composants* [Garlan 2000] et la *modélisation par objets* [Engels et Gregor 2000] [Smeda 2006].

La première approche, qui a émergé au sein de la communauté de recherche « *architectures logicielles* » décrit un système logiciel comme un ensemble de composants qui interagissent entre eux par le biais de liens simples ou complexes. Les chercheurs dans ce domaine ont permis d'établir les bases intrinsèques pour développer de nouveaux langages de descriptions d'architectures logicielles. L'architecture logicielle a permis, notamment, de prendre en compte les descriptions de haut niveau des systèmes complexes et de raisonner sur leurs propriétés à un haut niveau d'abstraction (protocole d'interaction, conformité avec d'autres architectures, etc.).

La seconde approche est devenue de facto un standard de description d'un système logiciel durant les dernières années. Avec l'unification des méthodes de développement objet sous la houlette du langage UML [Booch *et al.* 1998] [Booch 1994], cette approche est largement utilisée et bien appréciée dans le monde industriel.

Dans le cadre de cette thèse, nous nous intéressons d'une manière spécifique aux architectures logicielles à base de composants [Bass *et al.* 1998]. Ces dernières reposent sur une décomposition du système en un ensemble de composants (unités de calcul ou de stockage) qui communiquent entre eux par l'intermédiaire de connecteurs (unités d'interactions). Pour asseoir le développement de telles architectures, il est nécessaire de disposer de notations formelles et d'outils d'analyse de spécification architecturale. Les langages de description d'architecture (en anglais, les ADLs : *Architecture Description Languages*) constituent une bonne réponse. La communauté académique a proposé ces dernières années un nombre important d'ADLs pour décrire et raisonner sur les architectures logicielles.

Les deux approches de modélisation d'architectures logicielles citées précédemment ont plusieurs points en commun. Elles se basent sur les mêmes concepts que sont l'encapsulation, l'abstraction, la modularité et les interactions entre les entités conçues dans chaque approche. En termes d'architecture logicielle, en général, la similarité entre les deux approches est évidente. En termes d'intention, les deux approches ont pour but de réduire les coûts de développement des logiciels et d'augmenter la production des lignes de codes puisqu'elles permettent la réutilisation et la programmation à base d'objets et/ou composants. Ainsi, en tenant compte de l'importance accordée à l'approche architectures logicielles et la similarité entre ces deux approches, nous rejoignons David Garlan [Garlan 2000] [Garlan *et al.* 2002] sur le principe de « *réconciliation* » des besoins des deux approches. Par conséquent la tendance actuelle est l'utilisation conjointe des forts apports de ces deux approches pour décrire les architectures logicielles.

Problématique abordée

Le concept de connecteur a fait débat lors de développement d'architectures de systèmes logiciels complexes. Il a fallu du temps et plusieurs travaux pour pouvoir comprendre et converger vers le même sens en évoquant le mot « *connecteur* ». En revanche, s'il y a eu un accord sur le terme dans une approche, ceci n'est pas vraiment le cas dans les autres approches existantes. En effet, bien qu'il existe plusieurs travaux sur les interconnexions entre composants, le concept de connecteur reste mal défini. Il existe des considérations très différentes et même contradictoires concernant les connecteurs. Par exemple, certaines approches, comme les modèles de composants, ne définissent les interconnections entre les composants qu'avec des liaisons simples point-à-point. Ceci laisse une grande partie des communications à la charge des composants et limite ainsi leur réutilisation. D'autres approches les décrivent souvent comme des entités à part entière mais les réalisent à base de connexions simples ou sous forme de composants d'interactions. Il est ainsi difficile de préserver un connecteur comme une entité unique dans les différentes phases du processus de développement.

La problématique de la modélisation d'une application à base de composants, de connecteurs et de configurations est un point crucial. L'architecture logicielle permet d'obtenir une vision globale des éléments intervenant au sein d'une application. De ce constat évident du besoin lié à la description d'une architecture logicielle, sont nées plusieurs problématiques :

1- une faible prise en charge des connecteurs où très peu d'ADLs les considèrent comme entités de première classe au même niveau conceptuel que les composants,

2- la plupart des langages de description ne traitent pas le concept de configuration explicitement,

3- la description hiérarchique avec plusieurs niveaux de descriptions imbriquées est mal supportée par plusieurs ADLs,

4- peu d'importance est accordée à la relation de conformité entre une architecture et ses implémentations,

5- le problème de description d'architectures à hiérarchies multiples (structurelle, fonctionnelle, conceptuelle, et de méta-modélisation) est rarement traité dans les langages de description d'architectures.

Autant de problèmes qui sont pour le moment peu pris en compte dans les langages de description d'architecture. Dans cette optique, le but de cette thèse est de définir un modèle de description d'architecture dans lequel nous défendons l'existence des connecteurs et des configurations nous adoptons les approches qui les considèrent comme des entités de première classe, ce qui permet une meilleure séparation des responsabilités et une réutilisation importante. Cependant, nous révisons cette notion de connecteurs en donnant une définition plus précise. Pour cela, nous définissons un connecteur comme une entité d'architecture abstraite qui évolue dans un processus de développement pour se concrétiser. Il tient son existence par ses propriétés et non pas par ses services, en définissant des interfaces explicites. Dans ce modèle, nous considérons qu'un connecteur est indépendant de toute plateforme. Ceci donne au connecteur un vrai statut d'entité de première classe dans le processus de développement logiciel. Ainsi, notre travail possède, en outre, une vocation de classification des moyens de communication dans une architecture à base de composants.

Contributions

Concrètement, dans cette thèse nous abordons les architectures logicielles à base de composants selon trois axes représentants des points de vue différents de l'architecture logicielle :

1- L'aspect *conceptuel*, par le biais d'un méta-modèle pour la description d'architectures logicielles,

2- L'aspect *méthodologique*, pour développer une méthodologie à suivre dans la description d'architectures logicielles,

3- L'aspect *outillage*, où on définit un profil UML pour notre méta-modèle, afin de rapprocher les architectures logicielles développées au standard industriel (UML).

Les contributions principales de cette thèse sont :

- La séparation entre les interactions (connecteurs) et les calculs (composants) augmente la réutilisation des composants et des connecteurs. Pour parvenir à ce but, nous avons proposé d'expliciter les connecteurs en tant qu'entités de première classe pour traiter les dépendances complexes entre les composants et une nouvelle structure pour les connecteurs qui devra garantir un assemblage correcte des composants et des configurations dans une architecture donnée,
- Le développement d'une méthodologie de description des architectures logicielles à hiérarchies multiples, en utilisant plusieurs types de connecteurs permettant de décrire plusieurs hiérarchies comme support de raisonnement sur l'architecture logicielle,
- L'élaboration de configurations explicites et ce, dans le but de les rendre réutilisables et extensibles,
- La prise en charge du problème de l'évolution dans les architectures logicielles. Dans cette perspective, nous proposons deux vues de description d'architectures logicielles : une vue externe représentant le modèle logique d'architecture et une vue interne qui représente l'architecture de l'application instance du modèle d'architecture,
- Le développement d'une démarche d'élaboration d'architectures logicielles en se basant sur le modèle en Y (MY).

Ainsi, notre principale contribution est de proposer un méta-modèle de description d'architecture que nous appelons C3 (*Composant, Connecteur, Configuration*). Aussi, nous proposons une démarche d'élaboration d'une architecture logicielle qui permet de guider le concepteur dans son processus de modélisation d'architectures logicielles. Cette démarche basée sur un modèle, baptisé MY, décrit les concepts d'architectures logicielles selon trois branches : composant, connecteur et configuration. Ces trois branches représentent successivement tout ce qui est lié aux calculs, aux interactions et à la structure ainsi que la topologie du système décrit. Le modèle MY a quatre niveaux conceptuels : méta méta-architecture, méta-architecture, architecture et application. Ce modèle améliore la réutilisation des architectures logicielles en supportant une bibliothèque multi-hiérarchies pour les quatre niveaux conceptuels.

Enfin, nous revenons sur un travail développé au sein de notre équipe de recherche (MODAL) qui porte sur la méta-modélisation d'architectures logicielles dénommé MADL (*Meta Architecture Description Language*) qui peut être vu comme une base pour l'unification des langages de description d'architectures et qui peut être utilisé pour établir la correspondance entre les concepts des langages de description d'architectures et ceux d'UML.

Organisation du document

Cette thèse est composée de cinq chapitres :

Le chapitre 1 pose le lit de notre travail et propose un état du domaine de l'architecture logicielle, notamment dans ses aspects nécessaires pour ce travail à savoir les composants, les connecteurs et les configurations. Le chapitre débute sur une rétrospective de l'architecture logicielle depuis les premières intuitions des années 70 jusqu'à son éclosion en tant que discipline à part entière dans les années 90.

Malgré le gain en popularité de l'architecture logicielle ces dix dernières années, le vocabulaire est souvent mal utilisé et les objectifs mal compris. Ainsi, nous avons jugé utile de présenter en premier lieu les différentes approches de description d'architectures logicielles et de rappeler quelques définitions notables de l'architecture logicielle proposées dans la littérature et de revenir sur les concepts noyaux de composants, de connecteurs et de configurations qui ont profondément influencé notre travail.

Le chapitre se poursuit avec les principaux mécanismes opérationnels des ADLs et se termine par l'emploi d'une technique de méta-modélisation pour la description des composants, des connecteurs et des configurations pour parvenir à unifier la description des architectures logicielles.

Le chapitre 2 cherche à situer le contexte du travail à travers nos motivations. Dans premier temps, nous présentons les différentes caractéristiques liées à la modélisation des connecteurs et des configurations. Ensuite, nous introduisons les principaux langages de description d'architectures ainsi que les différentes écoles académiques qui ont contribués au développements des ADLs, à savoir l'école américaine, l'école européenne et l'école française. Le chapitre se poursuit avec une synthèse sur la prise en charge de ces écoles pour les concepts de connecteur et de configuration qui représentent les points angulaires de toute représentation architecturale. Une conclusion sur cette étude analytique sera présentée à la fin du chapitre.

Le chapitre 3 décrit notre méta-modèle proposé en réponse aux limitations identifiées dans les ADLs étudiés au chapitre précédent. Ce méta-modèle baptisé C3 à pour objectif d'aborder le problème de modélisation des interactions entre composants ainsi que celui de la modélisation de la composition et de la représentation hiérarchique des configurations. Pour y parvenir, nous développons de nouveaux types de connecteurs permettant la description multi-vue des architectures logicielles à base de composants. Nous introduisons aussi, dans ce chapitre, le concept d'architecture externe (*vue logique*) et celui du concept d'architecture interne (*vue physique*) ainsi que leurs interactions permanentes. Il est fait état également à la fin de ce chapitre, d'une démarche de représentation des concepts de C3 suivant le formalisme en « MY ».

Le chapitre 4 fait l'état d'une étude analytique sur les différentes possibilités d'utiliser le standard UML dans la description des architectures logicielles à base de composants. A la fin, nous avons opté pour faire des extensions sur le méta-modèle UML 2.0 en utilisant des

stéréotypes. Par la suite nous avons regroupé les différents stéréotypes avec les valeurs marquées et les contraintes OCL (*Object Constraint Language*) associées pour former un profil UML (baptisé C3-UML) pour notre méta-modèle C3. Dans la deuxième section de ce chapitre, nous présentons la notion de méta-architecture et également le méta-modèle des langages de description d'architectures logicielles (MADL pour : *Meta Architecture Description Language*) qui fait l'abstraction des concepts d'architectures logicielles comme les composants, les connecteurs et les configurations. La troisième section de ce chapitre est consacrée à la présentation d'une stratégie pour la projection des concepts C3 vers UML 2.0. Cette stratégie est basée sur la projection des méta-concepts de l'architecture logicielle vers le MOF (*Meta Object Facility*) en utilisant le méta-modèle MADL. Nous clôturons ce chapitre par des expérimentations que nous avons opérées sur le méta-modèle C3.

En guise de conclusion, nous faisons le bilan de ce travail, pour souligner très précisément nos principaux apports et contributions apportées par cette thèse. Ensuite, nous procédons à une critique de ce travail et proposons alors des extensions possibles.

Le résumé de la structure de la thèse est illustré selon le schéma suivant :

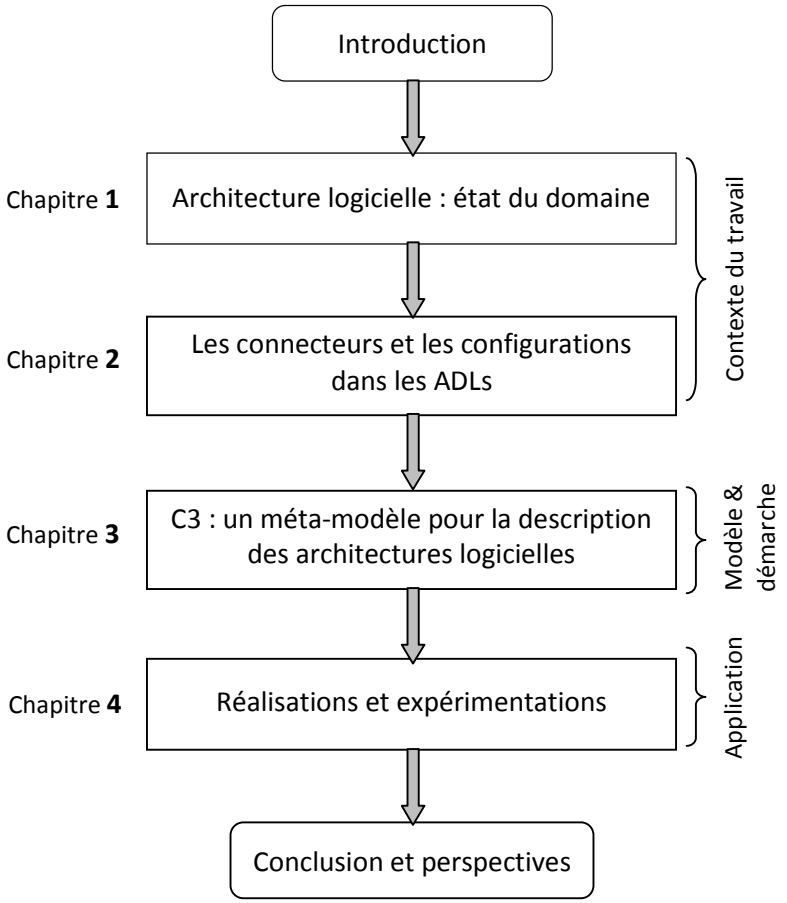


Figure I.1- Représentation schématique de la structure de cette thèse.

CHAPITRE 1

Architecture Logicielle : état du domaine

1.1 Introduction

L'architecture logicielle est une discipline récente du génie logiciel se focalisant sur la structure, sur les comportements ainsi que sur les propriétés globales d'un système. Elle s'adresse plus particulièrement à la conception de systèmes de grande taille. Durant de nombreuses années, l'architecture logicielle était décrite en termes de « *boîtes-et-lignes* ». Ce n'est qu'au début des années 90 que les concepteurs de logiciels se sont rendu compte du rôle déterminant que joue l'architecture logicielle dans la réussite du développement, de la maintenance et de l'évolution de leur système logiciel. Une bonne conception d'une architecture logicielle peut accoucher un produit performant qui répond aux besoins des clients et qui peut être modifiable et perfectible où de nouvelles fonctionnalités peuvent être facilement ajoutées; alors qu'une architecture inappropriée peut avoir des conséquences désastreuses qui peuvent aller jusqu'à l'arrêt du projet [Garlan 2000].

Les systèmes logiciels complexes exigent des notations expressives pour représenter leurs architectures logicielles. Deux principales approches se distinguent pour décrire et modéliser les systèmes logiciels et les intercommunications entre leurs entités : l'approche orientée objet, dite « *architecture logicielle à base d'objets* » [Brinkkemper *et al.* 1995] [Booch 1994] [Jacobson *et al.* 1992] [Martin 1993] et l'approche orientée composants appelée « *architecture logicielle à base de composants* » [Bass *et al.* 1998] [Garlan et Shaw 1993] [Shaw et Garlan 1996] [Kazman 2001] [Clements *et al.* 2003]. Chacune d'elles porte l'accent sur des aspects différents du développement logiciel en termes de niveau d'abstraction, de granularité et de mécaniques opératoires. Il n'y a pas de frontière nettement explicite entre les deux approches. En ce sens, il est possible de retrouver des points communs entre ces deux approches tant au niveau des motivations que des techniques et des méthodes. D'ailleurs, les concepteurs et les développeurs de systèmes à base de composants sont souvent tentés d'utiliser les modèles à objet comme support de spécification, de conception et d'implémentation [Garlan 2000] [Medvidovic *et al.* 2002].

Après la publication de l'article Perry et Wolf [Perry et Wolf 1992], nous constatons une forte augmentation de la recherche académique sur l'architecture logicielle. Cette recherche a été caractérisée par une variété de projets de différentes institutions, dont chacune a exploré une série de préoccupations à partir de sa propre perspective architecturale. Chaque projet a abouti à l'élaboration d'une notation et d'outils pour capturer l'architecture logicielle, avec la prise en compte d'une préoccupation particulière. Ces notations sont appelées langages de description d'architecture en anglais (*Architecture Description Languages* ou ADLs).

Darwin [Magee *et al.* 1995], *Wright* [Allen et Garlan 1997], *Rapide* [Luckham *et al.* 1995] *et C2SADEL* [Medvidovic *et al.* 1996] sont des exemples qui caractérisent le début de développement et la prolifération des ADLs. Ils partagent les mêmes concepts pour la modélisation de la structure à savoir les composants, les connecteurs, les interfaces et les configurations. Pour exprimer la structure de chacun de ces langages, la syntaxe est étrangement similaire, malgré le fait que ces langages ont été développés par des groupes indépendants dans différentes universités. Cependant, chaque langage comporte des fonctionnalités qui vont au-delà d'une simple description de la structure.

Dans ce chapitre, nous présentons les concepts de base des architectures logicielles, les principales approches de développement de langages de description d'architecture existantes, ainsi que leurs avantages et inconvénients. Nous présentons également le concept de méta-modélisation et une brève discussion sur le langage de méta-modélisation par objets (MOF : *Meta Object Facility*) et sur le langage de méta-modélisation par composants (AML : *Architecture Meta Language*).

1.2 Historique

Les concepts de bases des architectures logicielles, tels qu'ils sont connus aujourd'hui, remontent aux débuts de la discipline du génie logiciel. Les travaux pionniers de Dijkstra en 1968 sur la matière ont proposé une structuration impérative d'un système avant même de se lancer à écrire des lignes de code [Dijkstra 1968]. De même, les travaux de cet auteur mettent en évidence le besoin de la notion de « *niveau d'abstraction* » dans la conception de systèmes à grande taille. Bien que Dijkstra n'utilise pas le terme « *architecture* » dans l'ensemble de ses travaux, les résultats de ses travaux représentent une base pour la définition des architectures logicielles contemporaines.

On cite souvent Sharp [Buxton et Randell 1970] pour ses commentaires faits en 1969, en relation avec les travaux précédents de Dijkstra, lors de la conférence de l'OTAN de 1969 (*Software Engineering Techniques*) :

Sharp: "I think that we have something in addition to software engineering: something that we have talked about in small ways but which should be brought out into the open and have attention focused on it. This is the subject of software architecture. Architecture is different from engineering".

Pour illustrer « ses mots », Sharp donne l'exemple du OS/360 : « ce système représente un bon travail d'ingénierie puisqu'il a été très bien codé. Par contre, il est constitué d'un ensemble amorphe de programmes parce qu'il n'a pas été conçu par un seul architecte ». Les commentaires de Sharp résonnent encore bien 40 ans après. Il considère les travaux de Dijkstra comme une école d'architecture *"Dijkstra School of Architecture"* [Buxton et Randell 1970].

Au début des années 70, les travaux de Parnas ont introduit le concept de « *modularité* » qui propose d'améliorer la souplesse et le contrôle conceptuel du logiciel en réduisant le temps de développement des systèmes. Il a montré l'importance d'une bonne structuration des systèmes dans les étapes de conception dont il propose certaines idées afin d'atteindre un niveau de structuration adéquat [Parnas 1972].

Une différence importante entre ces deux auteurs est le fait que les notions introduites par Dijkstra se sont focalisées sur des aspects de programmation (*implémentation*) alors que les travaux de Parnas se sont centrés sur les niveaux de conception.

C'est incontestablement dans les années 90 que les architectures logicielles commencent à avoir un essor important. Les idées de base, telles que donner de l'importance aux décisions prises dans les premières étapes du développement du système ainsi qu'à la définition correcte de l'architecture logicielle, ont joué un rôle déterminant dans l'évolution de la discipline.

A l'heure actuelle, il n'existe toujours pas une définition précise et normalisée de l'architecture logicielle mais il en existe de nombreuses interprétations. Dans la section suivante, nous nous contenterons de présenter les définitions les plus influentes, classées par ordre chronologique.

1.2.1 Perry et Wolf (1992)

L'une des premières définitions formelles d'architectures logicielles, de Perry et Wolf [Perry et Wolf 1992], est restée l'une des plus perspicaces. Après avoir examiné les architectures dans d'autres disciplines (bâtiments, réseaux, mécanique), Perry et Wolf décrivent l'architecture logicielle comme étant un modèle constitué de trois composants : les éléments, la forme et le raisonnement.

Les *éléments* sont soit des éléments de traitements, soit des éléments de données ou des éléments de connexions.

La *forme* de l'architecture est définie en termes de propriétés des différents éléments, les relations entre ces éléments ainsi que les contraintes sur chaque élément.

Le *raisonnement* (*rational* en anglais) fournit le fondement de base de l'architecture en fonction des contraintes du système qui découle souvent des exigences du système.

1.2.2 Garlan et Shaw (1993)

Une définition très populaire de l'architecture logicielle a été avancée par Garlan et Shaw [Garlan et Shaw 1993] [Garlan et Perry 1995], qui est plus restrictive que la définition de Perry et Wolf. Garlan et Shaw ont proposé qu'une architecture logicielle pour un système spécifique soit représentée comme « un ensemble de composants de calcul – ou tout simplement de composants – accompagné d'une description des interactions entre ces composants – les connecteurs ».

Sur la base de cette définition, les auteurs ont utilisé le terme de style architectural pour désigner une famille de systèmes (c'est-à-dire d'architecture applicative) qui partagent un vocabulaire commun de composants et de connecteurs, et qui répondent à un ensemble de contraintes pour ce style. Les contraintes peuvent porter sur une variété de choses, notamment sur la topologie des connecteurs et des composants, ou sur la sémantique de leur exécution.

1.2.3 Bass, Clements et Kazman (1998)

Dans leur définition, Bass et al. [Bass *et al.* 1998] insistent sur les propriétés extérieurement visibles d'un composant qui définissent son comportement attendu. Ces propriétés traduisent les hypothèses que d'autres composants peuvent faire sur ce composant (par exemple, les services qu'il fournit, les ressources requises, ses performances, ses mécanismes de synchronisation). La qualification « extérieurement visible » exprime le pendant des détails que le composant encapsule.

Un composant est donc une unité d'abstraction dont la nature dépend de la structure considérée dans le processus de conception architecturale. Il peut représenter un service, un module, une bibliothèque, un processus, une procédure, un objet, une application, etc. Mais le comportement observable de chaque composant fait partie de l'architecture puisqu'il détermine ses interactions possibles avec d'autres composants.

1.2.4 La norme AINSI/IEEE Std 1471 (2000)

L'IEEE a défini une norme pour la description architecturale de systèmes à prédominance logicielle (*software-intensive systems*), l'IEEE 1471[1] [IEEE 2000]. L'IEEE 1471 a sciemment proposé une définition généraliste capable d'englober plusieurs interprétations. Surtout, la norme IEEE 1471 établit un cadre conceptuel (cf. Figure1.1) et un vocabulaire pour désigner des problématiques architecturales des systèmes. Elle comprend l'utilisation de plusieurs vues, de modèles réutilisables au sein des vues et la relation qu'entretient l'architecture avec le contexte du système (appelé aussi environnement). En se basant sur ce cadre conceptuel, l'idée est d'identifier et d'édicter des pratiques architecturales pertinentes, et de les incuber.

[1] A été également adopté comme standard ISO en 2006 et publiée en tant que ISO/IEC 42010 : 2007

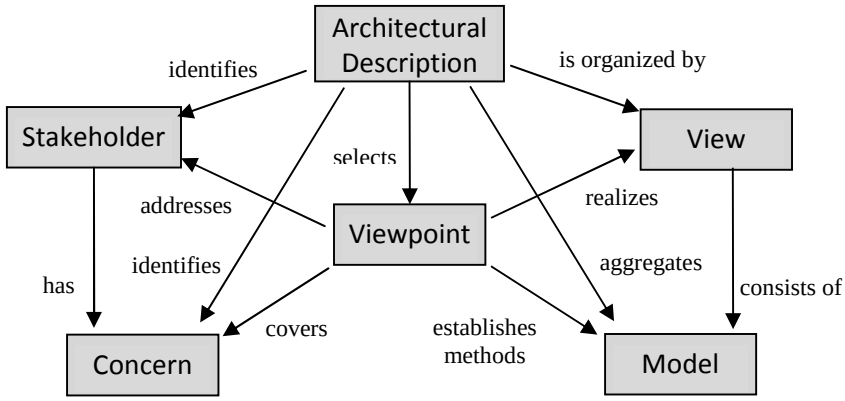


Figure 1.1 – Modèle conceptuel pour la description d'architecture logicielle (adapté de la norme 1471-2000).

1.3 Approches de description d'architectures logicielles

De nombreuses approches par composants industrielles ou académiques existent pour la conception ou la mise en œuvre des applications. L'hétérogénéité de ces approches et de leurs objectifs rend difficile une définition de la notion de composant logiciel. Cependant, un point commun que l'on trouve dans la plupart de ces approches réside dans la notion d'architecture logicielle définie comme une description abstraite du système, de sa décomposition en composants, de l'interface de ces composants et de leurs interactions.

Nous présentons dans les sections suivantes les deux principales approches pour décrire une architecture logicielle, à savoir, l'approche à base d'objets et l'approche à base de composants avec leurs avantages et inconvénients respectifs.

1.3.1 L'architecture logicielle à base d'objets

Dans cette section, nous présentons l'approche de modélisation par objet. Cette dernière est basée sur le paradigme « **objet** ». Elle est initialement prévue pour décrire les systèmes logiciels en se basant sur les définitions orientées objets. Principalement, il s'agit de décrire des systèmes comme une collection de classes (les entités à abstraire et l'encapsulation des fonctionnalités) qui peuvent avoir des objets « **instances** » et communiquent entre eux par des envois de messages.

1.3.1.1 Concepts de base de l'approche orientée objet

Le paradigme objet est principalement basé sur l'extraction et l'encapsulation des caractéristiques et des fonctionnalités des classes. Une classe peut être instanciée pour obtenir des objets. Un objet (*instance*) à des propriétés (*attributs*) et un comportement (*méthodes*). Les objets des différentes classes communiquent par l'intermédiaire d'envois de

messages. La Figure 1.2 montre les concepts de base de l'approche objet, que sont les classes, les objets et leurs relations. Cette figure montre également quelques mécanismes tels que l'héritage et la composition.

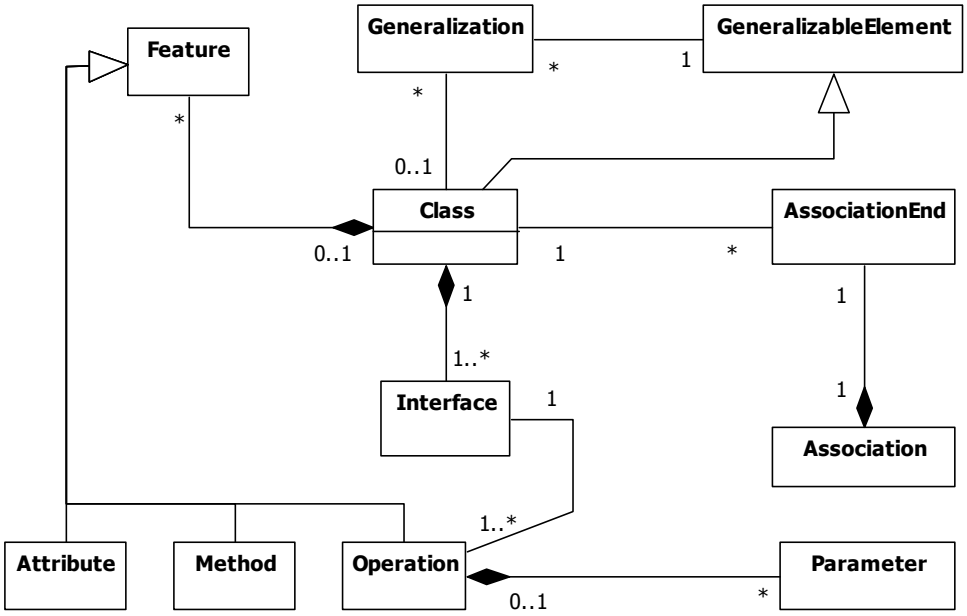


Figure 1.2- Les concepts de base de l'approche objet (extrait d'UML 1.1 [OMG 2003b]).

1.3.1.2 Les méthodes de modélisation par objet

La modélisation par objets (OOM : *Object Oriented Modeling*) consiste à créer des diagrammes, des textes de spécification et un code basé sur les concepts objets pour décrire un système logiciel [Engels et Gregor 2000]. Il s'agit d'examiner un problème selon de différents points de vue et de trouver une solution au problème. A l'image d'un ingénieur qui crée des schémas et des textes de spécification d'une nouvelle voiture, nous créons des diagrammes, des textes de spécification et un code pour décrire un nouveau système logiciel. Les langages de modélisation par objets sont des méthodes et des techniques pour analyser et représenter graphiquement les systèmes logiciels basés sur les concepts objet [Brinkkemper *et al.* 1995].

Il existe plusieurs méthodes de modélisation par objets, par exemple *Designing Object-Oriented Software* (DOOS) de Wirfs-Brock [Wirfs-Brock *et al.* 1990], *Object-Modeling Technique* (MOT) de Rumbaugh [Rumbaugh et al. 1991], *Object-Oriented Software Engineering* (OOSE) de Jacobson [Jacobson *et al.* 1992], *Objet-Oriented Analysis and Design* (OOAD) de Booch [Booch 1994]. Cependant, de nos jours, la plupart de ces méthodes sont intégrées dans UML [Booch *et al.* 1998], et de ce fait, ne sont plus pratiquées par les analystes.

1.3.1.3 Les avantages et les inconvénients de l'architecture logicielle à base d'objets

Les modèles à base d'objets présentent plusieurs avantages :

- ils se basent sur des méthodologies bien définies pour développer des systèmes à partir d'un ensemble de besoins,
- ils fournissent souvent une correspondance directe de la spécification à l'implémentation,
- ils sont familiers à une large communauté d'ingénieurs et de développeurs de logiciels,
- ils sont supportés par des outils commerciaux.

Cependant, on constate que l'approche objet souffre d'un certain nombre de lacunes par rapport aux systèmes à base de composants ou de services. Nous donnons ci-après les plus significatives [Pfister et Szyperski 1996] [Oussalah *et al.* 2005].

- l'approche objet a montré des limites importantes en termes de granularité et dans le passage à l'échelle. Le faible niveau de réutilisation des objets est dû en partie au fort couplage des objets. En effet, ces derniers peuvent communiquer sans passer par leur interface,
- la structure des applications objets est peu lisible (un ensemble de fichier),
- la plupart des mécanismes objets sont gérés manuellement (création des instances, gestion des dépendances entre classes, appels explicites de méthodes, …).

Par ailleurs, les approches objets :

- spécifient seulement les services fournis par les composants d'implémentation mais ne définissent en aucun cas les besoins requis par ces composants,
- fournissent un nombre limité de formes d'interconnexion (invocation de méthodes), rendant difficile la prise en compte d'interactions complexes,
- proposent peu de solutions pour faciliter l'adaptation et l'assemblage d'objets,
- prennent en compte difficilement l'aspect évolutif des objets (ajout, suppression, modification, changement de mode de communication,…).

Enfin, les méthodes objets :

- ne sont pas adaptées à la description de schémas de coordination et de communication complexes. En effet, elles ne se basent pas sur des techniques de construction d'applications qui intègrent d'une manière homogène des entités logicielles hétérogènes provenant de diverses sources,
- disposent de faibles supports pour les descriptions hiérarchiques, rendant difficile la description de systèmes à différents niveaux d'abstraction,

- permettent difficilement la définition de l'architecture globale d'un système avant la construction complète (implémentation) de ses composants. En effet les modèles à objets exigent l'implémentation de leurs composants avant que l'architecture ne soit complètement définie.

1.3.2 Architecture logicielle à base de composants

Les architectures logicielles à base de composants représentent une évolution logique des l'architectures logicielles à base d'objets. Elles décrivent les systèmes comme un ensemble de composants (unité de calcul ou de stockage) qui communiquent entre eux par l'intermédiaire de connecteurs (unité d'interactions). Leurs objectifs consistent à réduire les coûts de développement, à améliorer la réutilisation des modèles, à faire partager des concepts communs aux utilisateurs de systèmes et enfin à construire des systèmes hétérogènes à base de composants réutilisables sur étagères (COTS[2]). Pour asseoir le développement de telles architectures, il est nécessaire de disposer de notations formelles et d'outils d'analyse de spécifications architecturales. Les langages de description d'architecture (notés ADLs pour *Architectures Description Languages*) constituent une bonne réponse. La communauté académique a proposé ces dernières années un nombre considérable d'ADLs pour décrire et raisonner sur les architectures logicielles. Dans cette section, nous présentons les concepts de base de l'architecture logicielle à base de composants.

1.3.2.1 Concepts de base de l'approche orientée composant

Après les technologies objets qui ont modifié profondément l'ingénierie des systèmes logiciels améliorant ainsi leur analyse, leur conception et leur développement, nous abordons une nouvelle ère de conception de systèmes, « *l'orienté composant* », qui a émergé au sein de la communauté de recherche sur les « architectures logicielles » [Bass *et al.* 1998] [Garlan 1995] [Perry et Wolf 1992]. L'orienté composant consiste à concevoir et à développer des systèmes par assemblage de composants réutilisables à l'image par exemple des composants électroniques ou des composants mécaniques [Heineman et Councill 2001]. Plus précisément, il s'agit de :

- Concevoir et développer des systèmes à partir de composants préfabriqués, préconçus et pré-testés,
- Réutiliser ces composants dans d'autres applications,
- Faciliter leur maintenance et leur évolution,
- Favoriser leur adaptabilité et leur configurabilité pour produire de nouvelles fonctionnalités et de nouvelles caractéristiques.

[2] *COTS : Component-Off-The-Shelf* : acronyme utilisé pour désigner des composants prêts à l'emploi

Le terme composant existe dans le vocabulaire informatique depuis la naissance du génie logiciel. Il a cependant d'abord désigné des fragments de code alors qu'aujourd'hui, il englobe toute unité de réutilisation [Barbier *et al.* 2004].

A l'origine, l'approche par composants est fortement inspirée des composants de circuits électroniques. L'expérience gagnée dans cette discipline a largement contribué aux concepts de composants et de leur réutilisation. Concernant les composants électroniques, il suffit de connaître leur principe de fonctionnement et la façon dont ils communiquent avec leur environnement pour construire un système complet sans pour autant dévoiler les détails de leur implémentation. Comme le souligne Cox [Cox 1986] l'idée de circuits logiciels intégrés conduit à la notion d'éléments logiciels qui peuvent être connectés ou déconnectés d'un système plus complexe, remplacés et/ou configurés. Les constructeurs d'applications adoptent de plus en plus cette démarche. Il s'agit alors d'assembler à partir de composants logiciels de natures très diverses pour construire une application, on ne parlera alors plus de programmation d'applications mais plutôt de composition d'applications.

Aujourd'hui le développement d'applications à base de composants constitue une voie prometteuse. Pour réduire la complexité du développement, le coût de maintenance et accroître le niveau de réutilisabilité deux principes fondamentaux doivent être respectés : « *acheter plutôt que de construire* » et « *réutiliser plutôt que d'acheter* » [McIlroy 1968].

Ainsi, le concept de réutilisation de composants logiciels a été introduit par McIlroy en 1968 [McIlroy 1968] dans une conférence de l'OTAN consacrée à la crise du logiciel suite à un échec patent de la part de développeurs pour livrer des logiciels de qualité à temps et à un prix compétitif. L'idée est de mettre en place une industrie du logiciel pour accroître la productivité des développeurs et réduire le temps de mise sur le marché (*time-to-market*). Pour ce faire, les développeurs doivent construire des applications à partir de composants logiciels sur étagère plutôt que de les créer à partir de rien (*from scratch*). Ainsi, la construction d'applications pourrait se faire d'une manière plus rapide en assemblant des composants préfabriqués.

Cependant, plusieurs questions se posent dès lors qu'on aborde la problématique des composants. De fait, la situation des composants rappelle celle des objets au début des années soixante-dix : une syntaxe disparate qui renvoie à une sémantique ambiguë.

En effet, l'utilisation de l'approche « *composant* » dans le développement d'applications « *grandeur réelle* » révèle rapidement des interrogations. Par exemple : Quelle est la définition d'un composant ? Quelle est sa granularité ? Sa portée ? Comment peut-on distinguer et rechercher les composants de manière rigoureuse ? Comment peut-on les manipuler, les assembler, les réutiliser, les installer dans des contextes matériels et logiciels variant dans le temps, les administrer, les faire évoluer, etc.

1.3.2.2 Langages de description d'architectures (ADLs)

En accompagnement de la notion d'architecture logicielle, des formalismes sont apparus au cours des années 90 : les ADLs (*Architecture Description Languages*) ou langages de description d'architecture à base de composants qui sont utilisés pour décrire la structure comme un assemblage d'éléments logiciels. Cela est illustré par la Figure 1.3 où on voit un client et un serveur reliés par un appel de procédure à distance (*Remote Procedure Call*, RPC).

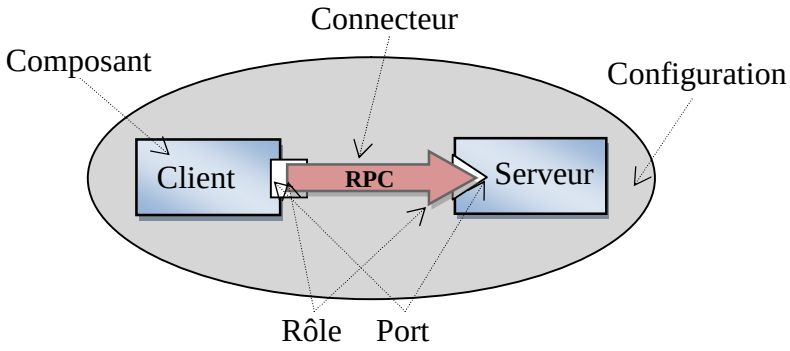


Figure 1.3- Assemblage d'éléments architecturaux pouvant être décrit à l'aide d'un ADL.

De nombreux ADLs ont été développés par le milieu académique, comme Wright, Darwin, ACME, UniCon, Rapide, Aesop, C2SADL, MetaH [Medvidovic et Taylor 2000]. De manière très générale, les entités manipulées par ces langages sont des éléments bien identifiés, suffisamment abstraits et les plus indépendants possibles les uns des autres. La spécification d'un assemblage de tels éléments constitue leur construction fondamentale. De ce point de vue, même s'ils affichent a priori une certaine adéquation avec la conception à base de composants lorsqu'elle vise la réalisation d'un système par assemblage de composants logiciels préexistants, nous rappelons qu'ils ont été originellement motivés par la maîtrise de la structure de plus en plus complexe des systèmes logiciels. Dans la section 1.4 nous décrivons l'ensemble des concepts que l'on trouve dans les ADLs.

1.3.2.3 Avantages et inconvénients de l'architecture à base de composants

Dans l'architecture logicielle à base de composants :

- les interfaces sont, en général, des entités de première classe décrites explicitement par des ports et des rôles,

- les interactions sont séparées des calculs et explicitement définies dans la plupart des ADLs [Medvidovic et Taylor 2000],

- les propriétés non-fonctionnelles sont prises en compte,

- les représentations hiérarchiques sont sémantiquement plus riches que de simples relations d'héritage. Elles permettent, par exemple, les représentations multi-vues d'une architecture [Garlan 2000],

- la description d'une architecture globale d'un système peut être spécifiée avant de compléter la construction de ses composants (implémentation),

- le niveau de granularité d'un composant ou d'un connecteur est plus élevé que celui d'un objet ou d'une association respectivement,

- un style architectural définit un vocabulaire de conception et indique un ensemble de contraintes portant sur ce vocabulaire. Les styles permettent à l'architecte de spécifier une tâche de conception aux domaines spécifiques et présentent des moyens pour améliorer l'analyse de systèmes.

Cependant l'architecture logicielle à base de composants :

- fournit seulement les modèles à un niveau élevé, sans expliciter comment ces modèles peuvent être reliés au code source. Comme le souligne Garlan [Garlan *et al.* 2000], de telles liaisons sont importantes pour préserver l'intégrité de la conception,

- reste un concept *ad-hoc* connu seulement par la communauté académique, actuellement, le monde industriel s'intéresse de plus en plus à cette discipline du génie logiciel,

- enfin, malgré la norme IEEE Std 1471-2000 [IEEE 2000], il n'y a pas un réel consensus vu que plusieurs notations et approches de description d'architectures logicielles sont proposées par la communauté d'architecture logicielle.

1.3.3 Synthèse sur les approches de modélisation d'architectures

Dans la section précédente nous avons présenté les différentes approches de description d'architectures logicielles. Nous avons pu faire ressortir un nombre de critères permettant de situer chaque approche. Dans le tableau 1.1 nous présentons ces différents critères et la position de ces approches par rapport à ces critères.

Critères	Architecture à base d'objets	Architecture à base de composants
Entité de base	objet	composant
Méthodologie	bien définie	pas de standard
Outils de support	outils commerciaux	outils plus académique
Familiarité	large communauté	communauté limitée
Correspondance (modèle→code)	directe	non explicite
Granularité	faible	élevé
Réutilisation	faible/niveau micro	forte/niveau composant
Evolution	difficile	facile
Abstraction	moyenne	élevé
Composition	il faut la programmer	composition native
Couplage	fort entre objets	faible entre composants
Lisibilité de la structure	peu lisible	très lisible
Adaptation à l'assemblage	peu de solutions	plusieurs solutions
Intégration d'entités	homogènes	homogènes/ hétérogènes
Schéma de communication	simple	complexe

Tableau 1.1- Synthèse des critères dans chaque approche de modélisation.

D'après cette étude, nous constatons que le concept d'ADL n'existe pas dans les architectures logicielles à base d'objets. Les ADLs sont définis uniquement dans les architectures à base de composants. Principalement, nos travaux de recherche sont focalisés sur les différents concepts relatifs aux ADLs. Par conséquent, dans le reste de cette thèse nous nous intéresserons très particulièrement aux architectures à base de composants.

1.4 Les concepts de base des ADLs

En ce qui concerne les architectures logicielles, la plupart des travaux récents publiés sont dans le domaine des langages de description d'architectures (ADLs). Un ADL est, selon Taylor et Medvidovic [Medvidovic et Taylor 2000], un langage qui offre des fonctionnalités pour la définition explicite et la modélisation de l'architecture conceptuelle d'un système logiciel, comprenant au minimum les concepts : de composant, de connecteur, d'interface et de configuration architecturale. Nous donnons dans ce qui suit une définition plus précise de ces concepts.

1.4.1 Composant

La notion de composant logiciel est introduite comme une suite à la notion d'objet. Le composant offre une meilleure structuration de l'application et permet de construire un système par assemblage de briques élémentaires en favorisant la réutilisation de ces briques [Shaw *et al.* 1995]. Le composant a pris de nombreuses formes dans les différentes approches de l'architecture logicielle. La définition suivante, globalement acceptée, illustre bien ses propriétés.

1.4.1.1 Définition d'un composant

Un *composant* représente le principal élément de calcul et de stockage des données dans un système. Chaque composant possède une interface, qui définit les points d'interaction entre cet élément et son environnement. La taille et la complexité d'un composant sont très variables. Un serveur, une base de données ou une fonction mathématique sont des exemples de composants. L'aspect clé de n'importe quel composant est qu'il peut être « *vu* » de l'extérieur seulement et via son interface uniquement par ses utilisateurs humains ou logiciels. Ainsi, il apparaît de l'extérieur comme une «*boîte noire*». Les composants logiciels sont donc la concrétisation des principes de génie logiciel de l'*encapsulation*, de l'*abstraction*, et de la *modularité*. Et cela a un certain nombre de conséquences positives sur la *composabilité*, la *réutilisabilité* et l'*évolutivité* des composants [Taylor *et al.* 2009].

Les modèles de composants hiérarchiques définissent le concept de composant composite, c'est-à-dire un composant contenant à son tour une configuration de composants et de connecteurs. L'interface d'un composant est un ensemble de points d'interaction entre le composant et le monde extérieur. Une interface d'un composant dans un ADL indique les services (messages, opérations et variables) que le composant fournit. Les ADLs doivent également fournir des moyens pour exprimer les besoins d'un composant, c'est-à-dire les services requis à partir d'autres composants dans l'architecture.

En effet, le terme *composant* est utilisé depuis la phase de conception jusqu'à la phase d'exécution d'une application, il prend donc plusieurs formes. Dans cette thèse, nous parlons de type de composant et instance de composant.

- *Type de composant* : un type de composant est la définition abstraite d'une entité logicielle. Nous utilisons le type de composant pour la description du modèle architectural.
- *Instance de composant* : une instance de composant est au même titre qu'une instance d'objet, une entité existante et s'exécutant dans un système. Elle est caractérisée par une référence unique, un type de composant et une implémentation de ce type. Nous utilisons les instances de composants pour la description des opérations d'une application.

1.4.1.2 Exemple de description d'un composant

L'exemple proposé concerne un système filtre de type PipeFilter permettant de lire un flux de caractères pour le transformer en flux contenant les mêmes caractères en lettres

capitales. Le schéma présenté dans la Figure 1.4 illustre ce système sous forme de diagramme.

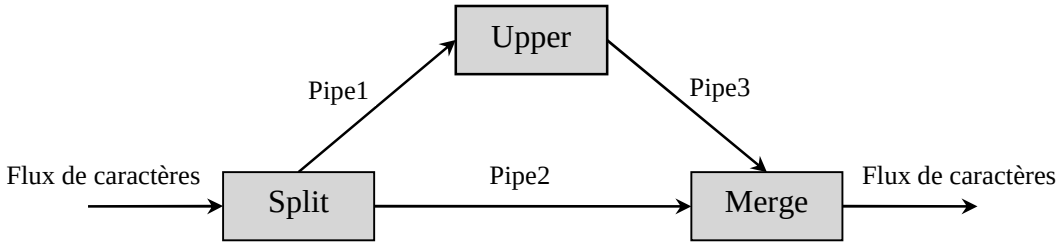


Figure 1.4 – Exemple de filtre.

Le système *Capitalize* comprend trois composants :

1- Le composant *Split* permet de récupérer le flux de caractères en entrée et de créer deux flux :
   - un flux identique à celui en entrée pour le composant *Upper* pour la conversion en lettres capitales,
   - un flux identique à celui en entrée pour le composant *Merge*.

2- Le composant *Upper* permet de transformer un flux de caractères en flux contenant les mêmes caractères en lettres capitales,

3- Le composant *Merge* permet de fusionner les deux flux.

En utilisant la description syntaxique d'ACME, le composant *Split* sera décrit comme le montre la Figure 1.5.

```
Component Split
{
    Port  Input      /* lire les données jusqu'à la fin
    Port  Left       /*sortir les données de manière filtrée
    Port  Right      /*sortir les données de manière filtrée
    Computation  /* lire continuellement les données à partir du port
                    Input puis les sortir soit dans le port Left soit dans le port Right */
}
```

Figure 1.5 – Description d'un composant dans l'ADL ACME.

L'interface du composant *Split* est formée de trois ports qui sont le port Input, Left et Right. La partie Calcul (*Computation*) du composant permet de mettre en rapport les trois ports.

1.4.2 Connecteur

Un autre aspect fondamental des systèmes logiciels est l'interaction entre les éléments constitutifs du système. Actuellement, plusieurs systèmes sont construits à partir d'un grand nombre de composants complexes, distribués à travers plusieurs sites éventuellement mobiles, et mis à jour dynamiquement sur de longues périodes. Dans de tels systèmes, assurer des interactions appropriées entre les composants est très important, et peut représenter un grand challenge pour les concepteurs et développeurs que les fonctionnalités individuelles des composants. En d'autres termes, les interactions dans un système sont devenues une principale préoccupation architecturale. Les connecteurs logiciels sont l'abstraction architecturale chargée d'effectuer et de réguler les interactions entre composants [Taylor *et al.* 2009].

1.4.2.1 Définition d'un connecteur

Un connecteur modélise l'interaction entre les composants et aussi les règles qui régissent cette interaction. Le connecteur peut représenter une interaction simple comme une invocation à un service, ou bien un protocole complexe, comme le contrôle d'un robot sur Mars par une station de contrôle au sol.

Chaque connecteur possède un type qui spécifie le modèle d'interaction d'une manière explicite et abstraite. Ce modèle peut être réutilisé dans différentes architectures. Un connecteur contient deux parties importantes qui sont un ensemble de rôles et la glu. Les rôles d'un connecteur permettent d'identifier les participants à l'interaction. La glu définit comment les rôles interagissent entre eux.

1.4.2.2 Exemple de description d'un connecteur

Si on prend l'exemple du système de filtre de type Pipe-Filter, les composants filtres sont liés entre eux par des tubes. Ces derniers fonctionnent de la même façon et obéissent aux mêmes règles. La Figure 1.6 illustre un connecteur Pipe définissant un tube.

```
Connector Pipe {
        Role Source   /*délivrer les données continuellement, signaler la
                        fin et fermer)*/
        Role Sink     /*lire les données continuellement, fermer au
                        moment de la signalisation de la fin des données */
        Glue          /*le rôle Sink reçoit les données dans le même ordre que celui
                        utilisé par le rôle source */
}
```

Figure 1.6 – Description d'un connecteur dans l'ADL ACME.

Les connecteurs sont tellement critiques et riches mais demeurent l'élément le moins pris en charge par les architectures logicielles. Dans ce qui suit, nous allons illustrer brièvement certains types de connecteurs qui sont peut être plus familiers.

Le type de connecteurs le plus simple et le plus utilisé est l'*appel de procédure*. Les appels de procédure sont directement implémentés dans le langage de programmation où ils permettent généralement l'échange synchrone de données et le contrôle entre composants. Un autre type de connecteur très commun est l'*accès aux données partagées*. Ce type de connecteur se manifeste dans les systèmes logiciels sous la forme de variables non locales ou de mémoires partagées. Les connecteurs de ce type permettent aux composants logiciels d'interagir par la lecture et l'écriture dans les zones partagées. Une classe importante de connecteurs dans les systèmes logiciels modernes concerne les connecteurs de *distribution*. Généralement, ces connecteurs encapsulent les bibliothèques réseaux des programmes d'interfaces (APIs) pour permettre aux composants d'interagir dans un système distribué.

1.4.3 Configuration

Les composants et les connecteurs sont composés d'une manière spécifique dans l'architecture d'un système donné pour accomplir l'objectif de ce système. Cette composition représente la *configuration* du système, également appelée *topologie*. Plusieurs définitions ont été proposées pour le concept de configuration. Dans ce qui suit, nous donnons la définition la plus adoptée.

1.4.3.1 Définition d'une configuration

Une configuration architecturale (ou simplement architecture ou système) est un graphe qui montre la façon dont un ensemble de composants sont reliés les uns aux autres par l'intermédiaire de connecteurs. Le graphe est obtenu en associant les ports des composants avec les rôles des connecteurs adéquats, en vue de construire l'application. Par exemple, les ports des composants de type « *Filter* » sont associés aux rôles de connecteurs de type « *Pipe* » à travers lesquels ils lisent et écrivent des flux de données (cf. Figure 1.8). L'analyse d'une configuration permet par exemple de déterminer si une architecture est « *trop profonde* », ce qui peut affecter la performance due au trafic de messages à travers plusieurs niveaux hiérarchiques, ou « *trop large* », ce qui peut conduire à trop de dépendances entre les composants.

1.4.3.2 Exemple de description d'une configuration

Si on prend l'exemple du système de filtre de type Pipe-Filter, la configuration qui correspond à l'architecture du système *Capitalisation* est illustrée par la Figure 1.7.

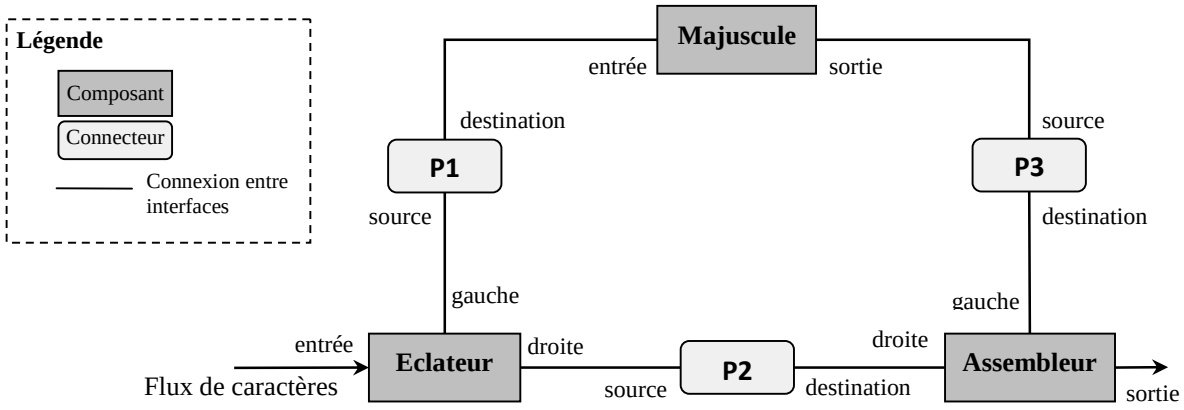


Figure 1.7 – Exemple d'une configuration du système *Capitalisation*.

Cette figure illustre la configuration architecturale de l'exemple capitalisation. Dans la figure *Eclateur* et *Assembleur* sont des exemples de composants tandis que P2 est un connecteur entre eux. La configuration illustrée dans le diagramme indique que ces deux composants peuvent interagir mutuellement via un connecteur. Toutefois, les informations affichées ne garantissent pas leur capacité réelle à interagir. En plus d'une connectivité valide dans une configuration, les composants doivent avoir des interfaces compatibles sinon ils seront une source d'incompatibilité architecturale. Notons que les interfaces ne sont pas représentées dans le diagramme de la Figure 1.7. La description syntaxique en utilisant ACME de la configuration capitalisation (*Capitalize*) est donnée par la Figure 1.8.

```
Configuration Capitalize
        Component  Split, Upper,  Merge  /*déclaration des composants
        Connector   Pipe    /* déclaration de connecteurs
        Instances          /*déclaration des instances
                Eclateur : Split
                Majuscule : Upper
                Assembleur : Merge
                P1, P2, P3 : Pipe
        Attachments   /*description des liens entre les composants et les connecteurs*/
                Eclateur.Left   to   P1.Source
                Majuscule.Input   to   P1.Sink
                Eclateur.Right  to  P2.Source
                Assembleur.Right  to  P2.Sink
                Majuscule.Output   to   P3.Source
                 Assembleur.Left   to   P3.Sink
End Configuration
```

Figure 1.8 – Description d'une configuration dans l'ADL ACME.

1.4.4 Interface

1.4.4.1 Définition d'une interface

Une interface constitue le « portail » des composants, des connecteurs et des configurations vers le monde extérieur. Elle spécifie les *ports* dans le cas des composants et des configurations, et les *rôles* dans le cas des connecteurs. Par exemple, si un composant implémente une API particulière, il posséderait probablement une interface indiquant que d'autres composants peuvent faire appel à cette API sur le composant cible.

Typiquement, les interfaces ont un identifiant unique, une description textuelle et une direction (un indicateur pour savoir si l'interface est fournie, requise, ou les deux). On peut également trouver une spécification de la sémantique de l'interface.

Une interface définit ainsi les engagements qu'un élément architectural[3] peut tenir et les contraintes liées à son utilisation. Les interfaces permettent également un certain degré de raisonnement, certes limité, portant sur la sémantique d'un composant.

Dans la plupart des ADLs, les interfaces d'un composant portent le nom de *ports*. Deux types de ports peuvent être distingués : les *ports services* (*fournis*), qui exportent les services de composants, et les *ports besoins* (*requis*), qui importent les services des composants.

Nous constatons donc que le concept d'interface est très important. Il peut être utilisé pour formaliser les connexions entre composants et/ou connecteurs. La communication ne peut se faire sans ces interfaces. Toutefois, certains ADLs ne les considèrent pas comme entité de première classe mais seulement comme une caractéristique technique basique [Szyperski 2002].

1.4.4.2 Type d'interfaces

Dans quelques ADLs les interactions entre les interfaces sont réalisées par des *attachments* et des *bindings*, comme c'est le cas dans ACME.

- *Attachment* (liaison) : représente la connexion entre un port de composant avec un rôle de connecteur, uniquement si ces derniers ont des directions compatibles de type requis/fournis.

- *Binding* (correspondance) : représente la connexion entre un port de configuration avec un autre port d'un sous-composant ou d'une sous-configuration (*élément interne*) uniquement si ces derniers ont des compatibles de type requis/requis ou fournis/fournis.

[3] Nous utilisons "*élément architectural*" pour référencer un composant, un connecteur ou une configuration

1.4.5 Style d'architecture

1.4.5.1 Définition d'un style architectural

Un style architectural est un moyen générique qui aide à l'expression des solutions structurelles des systèmes. Il comprend un vocabulaire d'éléments conceptuels (les composants et les connecteurs), impose des règles de configuration entre les éléments du vocabulaire (ensemble de contraintes) et véhicule une sémantique qui donne un sens (non ambiguë) à la description structurelle [Shaw *et al.* 1995] [Shaw et Garlan 1996].

Chaque style véhicule des propriétés logicielles spécifiques adaptées à des critères retenus pour un système. Ainsi, dans une organisation client-serveur :

- Un serveur représente un processus qui fournit des services à d'autres processus appelés clients ;
- Le serveur ne reconnaît pas à l'avance l'identité et le nombre de clients ;
- Le client connaît (ou peut trouver via un autre serveur) l'existence du serveur ;
- Le client accède au serveur, par exemple via des appels de procédures distants.

Le style d'une architecture permet de décrire un ensemble de propriétés communes à une famille de systèmes tels que les systèmes temps réel ou les applications orientées services. Il permet de décrire un vocabulaire commun en définissant un ensemble de types de connecteurs et de composants et un ensemble de propriétés et de contraintes partagées par toutes les configurations appartenant à ce style. La conformité d'un système à un style apporte plusieurs avantages, notamment pour : l'analyse, la réutilisation, la génération du code et l'évolution du système [Garlan *et al.* 1994] [Shaw et Garlan 1996] [Taylor *et al.* 1996].

1.4.5.2 Le style d'architecture C2

C2 est un exemple concret de style d'architecture défini dans [Taylor *et al.* 1996]. Ce style a été défini pour représenter les logiciels distribués. Une architecture suivant le style C2 présente les caractéristiques suivantes :

– les connecteurs permettent l'échange de messages entre composants,

– les composants réalisent des opérations et échangent des messages avec d'autres composants via deux interfaces appelées « top » et « *bottom* »,

– chaque interface considère un ensemble de messages qui peuvent être envoyés ou reçus. L'interface d'un composant ne peut être reliée qu'à un seul connecteur,

– deux types de messages sont permis : *request* (appel à un composant pour réaliser une opération) et *notification* (pour informer des opérations réalisées ou des changements d'état),

– un message *request* peut seulement être dirigé vers le haut de l'architecture, alors qu'un message *notification* peut seulement être dirigé vers le bas.

La Figure 1.9 illustre un exemple d'une architecture C2 pour un système de gestion de réunions. Les composants impliqués sont :

- *MeetingInitiator* qui s'occupe de l'organisation du calendrier des réunions,
- *Attendee* qui représente chacun des participants à une réunion préétablie,
- *ImportantAttendee* qui représente un type spécial de participant.

Trois connecteurs sont déclarés, chacun correspondant à un des composants. Certains messages sont envoyés depuis le *MeetingInitiator* vers les différents types de participants. D'autres messages sont adressés uniquement aux participants spéciaux.

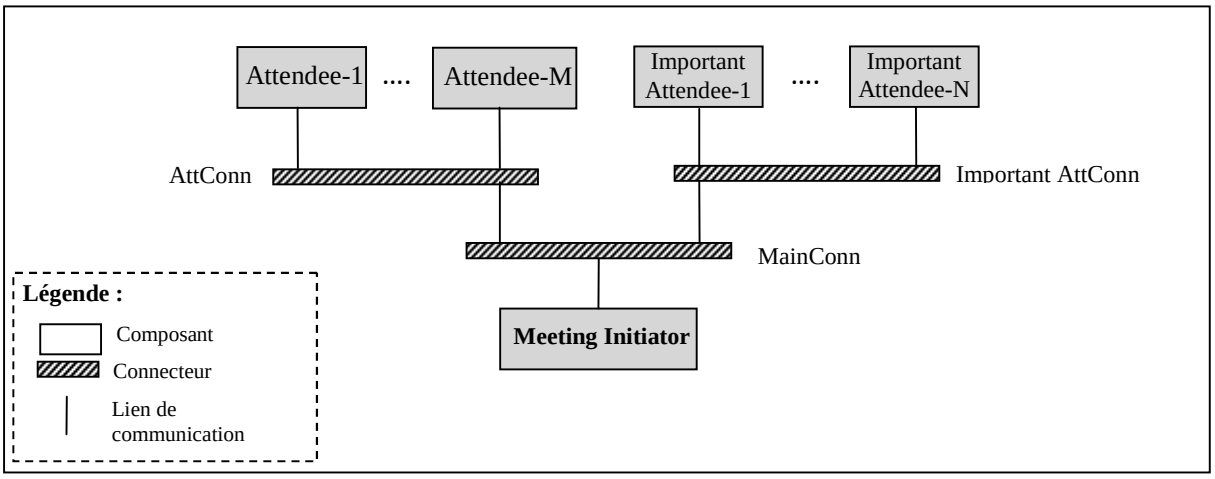


Figure 1.9- Style d'architecture C2 pour un système de gestion de réunions (adapté de [Taylor *et al.* 1996]).

C'est le composant *MeetingInitiator* qui entame les opérations en demandant les besoins de réunions aux composants *Attendees* et *ImportantAttendees*. Les composants concernés notifient au composant *MeetingInitiator* leurs besoins spécifiques, alors que celui-ci cherche à programmer des réunions en fonctions de leurs exigences. Dans la Figure 1.9, une bonne partie de l'information requise par cet exemple est implicite, ainsi un ADL s'avère nécessaire pour préciser les détails.

1.4.6 Concepts supplémentaires

Bien que les concepts précédents soient suffisants pour décrire une architecture logicielle, la majorité des ADLs y ont ajouté les concepts supplémentaires suivants :

1.4.6.1 Propriété

En supplément des éléments de haut-niveau précités, la plupart des architectures associent également des propriétés à leurs éléments constitutifs. Par exemple, pour une

architecture dont les composants sont associés à des tâches périodiques, les propriétés pourraient définir la périodicité, la priorité, et la consommation CPU de chaque composant. Les propriétés des connecteurs pourraient inclure la latence, le débit, la fiabilité, le protocole d'interaction, etc. Les propriétés peuvent être fonctionnelles ou non-fonctionnelles.

1.4.6.2 Type

Le type est au paradigme composant ce que la classe est au paradigme objet. Il permet l'encapsulation de fonctionnalités et des éléments internes en vu d'être réutilisés. Un type peut être instancié plusieurs fois dans une même architecture ou peut être réutilisé dans d'autres architectures. Les instances ont une structure et un comportement identiques à ceux de leurs types. Un système de type explicite facilite la compréhension et permet d'analyser les architectures [Garlan 1995]. On peut ainsi réutiliser des composants, des connecteurs et des configurations par le biais de types de composants, de types de connecteurs et de types de configurations. A cet égard, un style architectural peut être vu comme un type de configuration particulière.

1.4.6.3 Contrainte

Les contraintes expriment des restrictions sur les composants, les connecteurs et les configurations. Elles sont spécifiées au niveau des types pour s'appliquer sur les instances. Une contrainte sur un composant peut par exemple servir à borner les valeurs de ses propriétés. Un exemple de contrainte sur un connecteur est la restriction du nombre de composants qui interagissent à travers ce dernier. Une contrainte sur une configuration peut permettre de gouverner la typologie et le nombre d'instances autorisées. Les contraintes peuvent être spécifiées soit dans un langage de contraintes séparé, soit directement en utilisant les notations de l'ADL hôte.

1.5 Les mécanismes opérationnels des ADLs

Les architectures logicielles sont prévues pour décrire des systèmes logiciels à grande échelle qui peuvent évoluer dans le temps. Les modifications dans une architecture peuvent être planifiées ou non planifiées. Elles peuvent également se produire avant ou pendant leur phase d'exécution. Les ADLs doivent supporter de tels changements grâce à des mécanismes opérationnels. En plus, les architectures sont prévues pour fournir aux développeurs des abstractions dont ils ont besoin pour faire face à la complexité et à la taille des logiciels. C'est pourquoi les ADLs doivent fournir des outils de spécification et de développement pour pouvoir prendre en compte des systèmes à grande échelle susceptible d'évoluer. Aussi, pour améliorer l'évolutivité et le passage à l'échelle et pour augmenter la réutilisabilité et la compréhension des architectures, des mécanismes spéciaux doivent être pris en compte par les ADLs. Dans la suite nous présentons ces différents mécanismes.

1.5.1 L'instanciation

Avec les définitions données dans la section 1.4, les composants, les connecteurs et les configurations peuvent être instanciés plusieurs fois dans une architecture et chaque instance peut correspondre à une implémentation différente. L'instance d'un composant, d'un connecteur ou d'une configuration est un objet particulier, qui est créé avec le respect du plan donné par son type de composant, de connecteur ou de configuration. Toutes les instances d'un type de composant, de connecteur ou de configuration doivent inclure la structure définie par ce type et se comporter exactement de la même manière que ce dernier. Par exemple, si un ensemble de propriétés est définie pour un type particulier, chaque instance de ce type doit avoir les mêmes propriétés.

1.5.2 L'héritage et le sous-typage

L'héritage et le sous-typage sont deux manières différentes de réutiliser des modèles (de composants, de connecteurs ou de configurations). Alors que l'héritage permet la réutilisation du modèle lui-même, le sous-typage permet la réutilisation des constituants d'un modèle.

1.5.2.1 L'héritage

L'héritage est une représentation de la hiérarchie des abstractions, dans laquelle un sous-modèle hérite d'un ou de plusieurs modèles. Typiquement, le sous-modèle augmente ou redéfinit la structure et le comportement existant de son ou de ses modèle(s). L'héritage est un mécanisme puissant pour l'évolution car il permet à un modèle hérité d'être modifié en ajoutant, en supprimant ou en changeant sa structure [Oussalah *et al.* 2005]. Une relation d'héritage qui ressemble à celle disponible dans les langages à objets serait très acceptable dans les langages à composants. Pour permettre la prise en compte de systèmes sophistiqués et complexes, l'héritage dans les systèmes à base de composants est sélectif et dynamique. L'aspect sélectif permet de choisir les éléments à hériter et de les combiner (exemple, hériter l'interface d'un composant mais pas son implémentation, fusionner deux interfaces, etc.). L'aspect dynamique permet à des sous-modèles de modifier ce qu'ils ont hérité. L'héritage dans les ADLs peut également être multiple, où un sous-modèle hérite de plusieurs modèles.

1.5.2.2 Le sous-typage

Le sous-typage peut être défini par la règle suivante : un type X est un sous-type d'un type Y si les valeurs du type X peuvent être utilisées dans n'importe quel contexte où le type Y est prévu sans présenter d'erreurs. Dans [Medvidovic *et al.* 1999], les auteurs ont distingué trois relations de sous-typage différentes dans les architectures logicielles : le sous-typage d'interface, le sous-typage de comportement et le sous-typage d'implémentation. Le sous-typage d'interface exige que pour qu'un composant C1 soit un sous-type d'interface d'un autre composant C2, il faut que C2 spécifie au moins les interfaces fournies et au plus

les interfaces requis de C1. Le sous-typage de comportement exige que chaque opération fournie du super-type ait une opération fournie correspondante dans le sous-type. Enfin, le sous-type d'implémentation peut être établi avec un contrôle syntaxique si les opérations du sous-type ont des implémentations identiques que les opérations correspondantes du super-type [Oussalah *et al.* 2005].

1.5.3 La composition

Les architectures sont nécessaires pour décrire des systèmes logiciels à différents niveaux de détails, où les comportements complexes sont soit explicitement représentés, soit encapsulés dans des composants et des connecteurs. Un ADL doit pouvoir prendre en compte le fait qu'une architecture entière devienne un simple composant dans une autre architecture plus complexe. Par conséquent, la prise en compte de la composition est cruciale. Cependant, ce mécanisme exige que les composants aient des interfaces bien définies puisque leurs implémentations sont cachées. Comme nous le verrons dans le chapitre suivant, plusieurs ADLs utilisent ce mécanisme pour définir des configurations, où les systèmes sont définis comme des composants composites qui sont constitués de composants et de connecteurs.

1.5.4 La généricité

La généricité se rapporte à la capacité de paramétrer des types. L'instanciation ne fournit pas de services pour paramétrer des types. Souvent, des structures communes dans les descriptions de systèmes complexes sont amenées à être spécifiées à plusieurs reprises. Bien que l'héritage et la composition permettent de réutiliser le code, cela ne permet pas de résoudre tous les besoins de la réutilisation. Les types génériques permettent de réutiliser le code source en fournissant au compilateur la manière de substituer le nom des types dans le corps d'une classe. Ceci aide à la conception et à l'utilisation de bibliothèques de composants et de connecteurs. Ces dernières constituent des outils importants pour le développement rapide et efficace de logiciels à base de composants.

1.5.5 Le raffinement et la traçabilité

L'argument le plus souvent évoqué pour définir et utiliser les ADLs est qu'ils sont nécessaires pour établir le lien entre les diagrammes informels de haut niveau de type « boîtes-et-lignes » et les langages de programmation qui sont considérés comme des langages de bas niveau. Comme les modèles architecturaux peuvent être définis à différents niveaux d'abstractions ou de raffinements, les ADLs fournissent aux développeurs et aux concepteurs des outils expressifs et sémantiquement riches pour les spécifier. Les ADLs doivent également permettre une traçabilité des changements à travers ces niveaux de raffinement. Notons que le raffinement et la traçabilité sont les mécanismes les moins pris en compte par les ADLs actuels.

1.6 Architecture et méta-architecturation

A travers notre étude des ADLs, nous constatons que la spécification d'une architecture peut passer par plusieurs niveaux de modélisation. Dans cette section, nous appliquons les quatre niveaux de modélisation proposés par l'OMG à l'architecture logicielle. En effet, la technique de méta-modélisation peut reposer sur une approche objet (comme celle de l'OMG), ou bien sur une approche composant. On parlera aussi et respectivement de méta-modélisation par objets et de méta-modélisation par composants. Dans cette thèse, nous nous intéressons à la méta-modélisation par composants dont le résultat est une hiérarchie à différents niveaux d'architectures, qui passe de la définition d'une méta-méta-architecture à une application réelle à travers les différents niveaux intermédiaires.

1.6.1 Méta modélisation

Avant d'étudier ce qu'est un méta-modèle, il semble logique de s'intéresser d'abord à ce qu'est un modèle. Minsky propose la définition suivante [Minsky *et al.* 1968] : un objet (A^*) est un modèle d'un objet A pour un observateur O dans la mesure où O peut utiliser (A^*) pour répondre aux questions qu'il se pose sur A. dans ce cas, O utilisera le modèle (A^*) comme support de raisonnement. L'action de modélisation permet donc de passer de l'objet du modèle (A dans la définition de Minsky) vers le modèle (A^*), à travers un principe d'abstraction visant à simplifier la représentation de l'objet à modéliser. Il est possible de poursuivre ce travail d'abstraction pour créer des méta-modèles. La création d'un méta-modèle revient à construire une représentation d'un modèle, toujours dans le but de fournir un support de raisonnement. L'action de méta-modélisation permet donc de passer d'un modèle (A^* dans la définition de Minsky) vers le méta-modèle (A^{**}).

Comme le résume la Figure 1.10, l'objet (A^{**}) ne peut pas être obtenu directement par modélisation de l'objet (A), mais bien par la modélisation de l'objet (A^*). Cette relation d'ordre dans l'activité de modélisation fait apparaitre des niveaux de modélisations distincts.

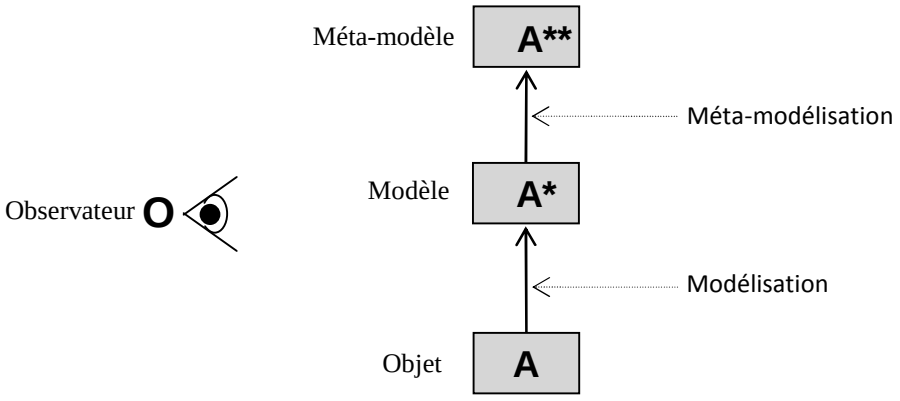


Figure 1.10–Activité de modélisation et de méta-modélisation dans les termes de Minsky.

1.6.2 Méta-modélisation par composant

La principale notation pour la méta-modélisation par objet est le MOF (*Meta Object Facility*) [OMG 2002a]. L'OMG a défini deux variantes de MOF : EMOF (MOF essentiel) et CMOF (MOF complet). Le but du MOF est de définir un langage unique et standard pour décrire des méta-modèles. Il est constitué d'un ensemble relativement restreint (bien que non minimal) de concepts « objets » permettant de modéliser ce type d'information. Par exemple, UML est l'un des méta-modèles décrit en utilisant le MOF. D'autres notations pour la méta-modélisation objet existent. Parmi elles on peut citer KM3 [Jouault et Bézivin 2006], ECORE [Budinsky *et al.* 2008] ou Kermeta [Muller *et al.* 2005].

Très peu de travaux ont été conduits dans la technique de méta-modélisation composant. AML (*Architecture Meta-Language*) [Wile 1999] est une première tentative de proposition d'un socle pour fournir aux ADLs une solide base sémantique. Il définit seulement les déclarations des trois constructions de base: les éléments, les types et les relations. MADL (*Meta Architecture Description Language*) [Smeda 2006] est une seconde tentative proposée par notre équipe. MADL est l'équivalent du MOF pour l'architecture logicielle. MADL inclut les concepts et les mécanismes des langages de description d'architecture, et conserve l'instanciation, l'héritage et la composition, issues de l'orientation objet [Smeda *et al.* 2008].

En pratique, un méta-métamodèle détermine le paradigme utilisé dans les modèles qui sont construit sur sa base. De ce fait, le choix d'un méta-méta-modèle est un choix important où la question de l'objet ou du composant comme une unité de construction de base est posée. En choisissant MADL [Smeda *et al.* 2008], nous optons pour le composant et nous identifions une hiérarchie de modèles architecturaux, comme le montre le Tableau 1.2. Les quatre niveaux de modélisation d'une architecture sont détaillés ci-après.

	Modélisation par objets	Modélisation par composants
M3 – Niveau méta-méta-modèle	MOF	MADL
M2 – Niveau méta-modèle	UML, CWM, SPEM, etc.	ACME, SOFA, ADL Fractal,…
M1 – Niveau modèle	Modèles UML	Architectures
M0 – Niveau instance	Informations réelles (Objets)	Applications

Tableau 1.2 – Hiérarchie selon une méta-modélisation par objets et par composants.

1.6.3 Niveaux de modélisation d'une architecture

Nous identifions quatre niveaux de modélisation dans les systèmes : le niveau méta-méta-architecture, le niveau méta-architecture, le niveau architecture et le niveau application. Nous schématisons ces niveaux sous forme d'une « *architecture pyramidale* » présenté dans

la Figure 1.11, où chaque étage de la pyramide se conforme à l'étage immédiatement supérieur.

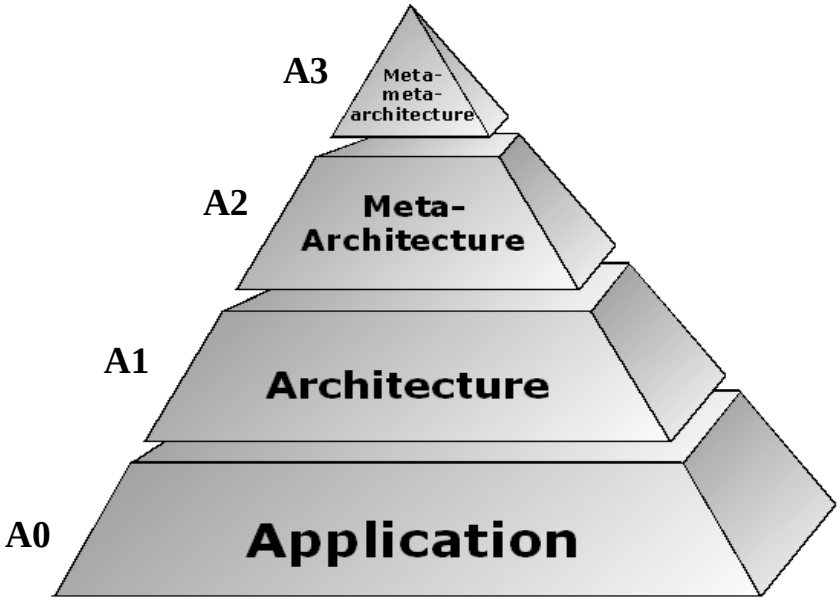


Figure 1.11 – Les quatre niveaux d'architecture que nous proposons.

*Le niveau méta-méta-architecture (*A3*)* : est le niveau le plus abstrait qui fournit les éléments minimaux de modélisation d'une architecture. Une méta-méta-architecture se conforme à elle-même (*i.e.*, s'auto définie). Les concepts de base d'un méta ADL sont définis à ce niveau.

*Le niveau méta-architecture (*A2) : fournit les éléments de modélisation de base pour un langage de description d'architecture (ADL) : composant, connecteur, configuration, port, rôle, etc. Ces concepts de base permettent de définir différentes architectures. Les méta-architectures se conforment au méta-méta-architecture fixée à l'avance. Dans le cadre d'une relation de conformité, chaque élément de (A2) est associé à un élément de (A3).

Le niveau architecture (A1) : à ce niveau, plusieurs types de composants, de connecteurs et de configurations sont décrits. Les architectures se conforment aux méta-architectures (ADLs), donc chaque élément de (A1) est associé à un élément de (A2).

Le niveau application (A0) : est le niveau où les unités d'exécution sont localisées. Une application est vue comme un ensemble de composants, de connecteurs et de configurations. Les applications sont conformes au niveau architecture. Chaque élément de (A0) est associé à un élément de (A1).

1.7 Conclusion

Nous avons présenté dans ce chapitre le contexte global dans lequel s'inscrit cette thèse : "*le domaine des architectures logicielles à base de composants*". Nous avons montré comment l'architecture logicielle s'est progressivement retrouvée sur le devant de la scène et les nouvelles pratiques de l'ingénierie qui sont apparues avec elle. Pour mieux comprendre ce que recouvre l'architecture logicielle, nous avons synthétisé les définitions notables proposées dans la littérature. Nous avons également consacré une section à la notion de langage de description d'architecture, que nous considérons comme un ingrédient clé du développement centré architecture. Puis, nous avons abordé les travaux sur les langages de description d'architectures, sur la base de leurs concepts communs. Nous avons clôturé ce chapitre par la description de l'architecture logicielle à travers ses différents niveaux de modélisation.

Les leçons retenues de ces vastes travaux sur les architectures logicielles est que l'accroissement de la complexité impose de monter en abstraction afin de garder le contrôle, le pouvoir de comprendre et de communiquer au sein d'un système. En plus, la capitalisation et la réutilisation des expériences passées via des solutions éprouvées sont des techniques phares pour guider et améliorer les activités des architectures. Nous sommes convaincus de la nécessité de prendre appui sur la méta-modélisation pour traiter correctement la problématique des langages de description d'architecture.

Les systèmes logiciels complexes nécessitent des notations expressives pour représenter leurs architectures logicielles. Les approches orientées objets, orientées composants et orientées services représentent à l'heure actuelle les meilleures réponses pour décrire les architectures logicielles de tels systèmes [Garlan 2000]. Chacune de ces approches se concentre sur certains aspects de l'architecture logicielle, les aspects fonctionnels pour la modélisation par objets, les aspects fonctionnels et non-fonctionnels pour la modélisation par composants et les aspects services pour la modélisation par services.

Dans ce chapitre, nous avons présenté ces approches d'architectures logicielles, leurs principaux concepts, leurs descriptions, leurs avantages et leurs inconvénients. On peut noter que la coexistence des qualités des ces approches sera très profitable pour l'architecture logicielle. Nous avons clôturé ce chapitre par un résumé sur le concept de méta-modélisation dans les approches orientées objets et composants, le langage de l'OMG pour la méta-modélisation par objets (MOF), et le langage AML qui est, apparemment, la seule approche de méta-modélisation par composants.

La modélisation des interactions et des configurations varie d'un ADL à un autre. Ceci dépend des objectifs et des préoccupations fixées pour chaque ADL. Dans le prochain chapitre nous présenterons une étude analytique ainsi qu'une comparaison entre les ADLs les plus représentatifs et les plus utilisés avec un focus sur les concepts de connecteur et de configuration.

CHAPITRE 2

Les connecteurs et les configurations dans les ADLs

2.1 Introduction

L'augmentation de la taille et la complexité des systèmes logiciels rendent de plus en plus difficile leur compréhension et leur évolution. Pour faire face à ce problème de complexité, des approches sophistiquées sont nécessaires pour décrire l'architecture de ces systèmes. La description architecturale est beaucoup plus évidente comme activité de conception et d'analyse dans le développement d'un logiciel. Comme nous l'avons mentionné dans le chapitre précédent, l'architecture d'un système logiciel peut être décrite en utilisant un langage de description d'architecture (ADL[1]) ou un langage de modélisation par objets. Ainsi, la communauté scientifique a proposé de nombreux langages de description d'architectures pour décrire les architectures de systèmes logiciels d'une part et d'autre part, le monde industriel a étendu la version initiale du langage UML[2] pour prendre en compte certains concepts architecturaux qui n'étaient pas encore disponibles dans UML.

Les langages de description d'architectures offrent une grande diversité de notations. Néanmoins les concepts de composant, de connecteur et de configuration sont généralement, considérés comme essentiels dans chaque langage. Un composant peut être défini comme étant une unité destinée à être assemblée et à fonctionner avec d'autres composants. L'architecture logicielle et matérielle d'un système peut être décrite comme une hiérarchie de composants. Les connecteurs permettent de spécifier les interactions entre ces composants.

Les premiers travaux réalisés pour décrire les architectures logicielles ont été menés dès le début des années 90. Depuis, un nombre important d'ADLs a été proposé pour la modélisation des architectures. Parmi ces langages nous trouvons ceux qui sont orientés vers un domaine bien particulier (DSL pour *Domain Specific Language*) et ceux qui sont indépendants du domaine *« General Purpose Language »*. Aussi, dans ces deux catégories, il y a des langages qui sont extensibles et d'autres qui ne le sont pas, comme nous trouvons des

[1] *ADL pour Architecture Description Language*
[2] *UML pour Unified Modeling Language*

langages qui ont une syntaxe et une sémantique formelle et d'autres qui ont uniquement une syntaxe rigoureuse avec une sémantique très faible. Il faut préciser qu'il existe plusieurs définitions pour les langages de description d'architectures mais la plus largement adoptée est celle proposée par Medvidovic et Taylor [Medvidovic et Taylor 2000].

« Les ADLs émergent comme des solutions à bases d'une notation textuelle ou graphique, formelle ou semi-formelle permettant de spécifier des architectures logicielles ».

Les ADLs permettent la définition d'un vocabulaire précis et commun pour les acteurs devant travailler autour de la spécification liée à l'architecture. Ils spécifient les composants de l'architecture de manière abstraite sans entrer dans leurs détails d'implémentation. Ils définissent de manière explicite les interactions entre composants d'un système, fournissent des outils pour aider les concepteurs à décrire l'aspect structurel et éventuellement comportemental d'un système.

Dans ce chapitre nous nous focalisons beaucoup plus sur les concepts noyaux de notre travail à savoir les connecteurs et les configurations. Les sections 2.2 et 2.3 nous introduisons respectivement les caractéristiques importantes à prendre en considération dans la modélisation des connecteurs et des configurations. Pour illustrer ces travaux la section 2.4 se focalise les principaux langages de description d'architecture qui ont largement inspiré nos travaux. Les sections 2.5 et 2.6 présentent respectivement des synthèses sur la modélisation des connecteurs et des configurations. La section 2.7 conclut ce chapitre.

2.2 Modélisation des connecteurs

Les connecteurs sont des blocs de constructions architecturaux utilisés pour modéliser les interactions entre les composants et les règles qui régissent ces interactions. Ils correspondent aux lignes dans les descriptions de type « *boîtes-et-lignes* ». Ils sont des entités architecturales qui relient des composants et agissent en tant que médiateurs entre eux. Contrairement aux composants, les connecteurs peuvent ne pas correspondre à des unités de compilation. Les modèles de connecteurs, dans les différents ADLs, existent sous différentes formes et avec différentes appellations. Nous pouvons les classer en trois grandes catégories :

- Les connecteurs *implicites* sont des connecteurs qu'on ne peut pas les instancier ; ils sont représentés en ligne dans la description d'architectures et ils ne peuvent être ni nommés ni typés,
- Les connecteurs *explicites* sont des connecteurs nommés et typés et par conséquent, nous pouvons les instancier,
- Les connecteurs *prédéfinis* sont des connecteurs explicites mais figés ; le concepteur ne peut pas intervenir pour les modifier.

Par exemple, dans ACME [Garlan *et al.* 1997], Aesop [Garlan 1995], C2 [Medvidovic *et al.* 1996], SADL [Moriconi *et al.* 1997], UniCon [Shaw *et al.* 1996] et Wright [Allen et

Garlan 1997] le modèle de connecteur est explicite et appelé *connecteur*, le connecteur est spécifié par un protocole. Weaves [Gorlick et Razouk 1991] modélise explicitement les connecteurs mais il les appelle *services de transport*. Par contre les *connexions* de Rapide [Luckham *et al.* 1995], *Koala* [Ommering 2002] et MetaH [Vestal 1996], les *binding* de Darwin [Magee *et al.* 1994] [Magee *et al.* 1995] et les liaisons de Fractal [Bruneton *et al.* 2003] [Bruneton 2004] sont modélisés d'une manière implicite (*in-line*) et ne peuvent en aucun cas être nommés, sous-typés ou réutilisés (ces connecteurs ne sont pas des entités de première classe).

2.2.1 Caractéristiques des connecteurs

Les connecteurs sont généralement définis à travers les six caractéristiques suivantes : l'interface, le type, les contraintes, l'évolution et les propriétés non-fonctionnelles.

2.2.1.1 Interface

Afin de permettre une connexion correcte entre composants et d'assurer leur interaction dans une architecture, un connecteur doit exporter les services qu'il prévoit pour rendre possible le raisonnement sur les configurations d'architectures. Par conséquent, l'interface d'un connecteur est un ensemble de points d'interaction entre lui-même et les composants qui lui sont attachés. Deux types d'interfaces existent : *l'interface requise* et *l'interface fournie*. En général, les seuls ADLs qui modélisent les connecteurs comme des entités de première classe supportent une spécification explicite des interfaces de connecteurs. La plupart de ces ADLs modélisent les interfaces de composants et de connecteurs de la même manière, mais avec des références différentes.

2.2.1.2 Type

Les types de connecteurs sont des abstractions qui encapsulent la communication entre composants. Au niveau architecture les interactions doivent être caractérisées par des protocoles simples ou complexes. Rendre ces protocoles réutilisables, dans une architecture ou inter-architectures, cela nécessite que les ADLs modélisent les connecteurs comme des types. Seuls, les ADLs qui modélisent les connecteurs comme des entités de première classe distinguent entre les types de connecteurs et leurs instances.

2.2.1.3 Contraintes

Les contraintes dans les connecteurs assurent le respect des protocoles d'interactions, permettent d'établir des dépendances intra-connecteurs et font respecter les limites de leurs utilisations. Un exemple d'une contrainte simple est de faire respecter une restriction sur le nombre de composants qui interagissent par le biais du connecteur.

2.2.1.4 Evolution

L'évolution d'un connecteur est défini comme étant la modification de (ou d'un sous-ensemble de) ses propriétés ; par exemple, l'interface, la sémantique ou les contraintes sur les deux propriétés précédentes. Les interactions entre composants dans une architecture donnée sont gérées et régulées par des protocoles complexes qui sont potentiellement changeants et extensibles. De plus, les composants individuels et leurs configurations évoluent. Les ADLs peuvent accommoder cette évolution en modifiant ou en raffinant les connecteurs existants avec des techniques telles que le filtrage d'informations incrémentales, sous-typage et le raffinement.

2.2.1.5 Propriétés non-fonctionnelles (PNF)

Les propriétés non-fonctionnelles d'un connecteur ne sont pas entièrement déduites de la spécification de sa sémantique. Elles représentent les besoins nécessaires pour une implémentation correcte des connecteurs. La modélisation des propriétés non-fonctionnelles permet la simulation du comportement en exécution, l'analyse des connecteurs et la sélection correcte des connecteurs sur étagère *« OTS Connectors*[3] *»*.

2.3 Modélisation des configurations

Les configurations architecturales ou les topologies sont des graphes connectés de composants et de connecteurs qui décrivent la structure architecturale d'un système. Une configuration est nécessaire pour déterminer si les composants appropriés sont connectés, leurs interfaces sont compatibles, les connecteurs permettent une communication correcte et leur sémantique combinée correspond au comportement global souhaité.

Le besoin de modélisation explicite des configurations distingue les ADLs de certains langages de conception de haut niveau. Les langages existants qui sont parfois appelés ADLs peuvent être regroupés en trois catégories selon la façon dont ils modélisent les configurations.

- *Langages à configurations implicites* : modélisent implicitement les configurations à travers les informations d'interconnexions qui sont réparties entre les définitions de composants et de connecteurs,
- *Langages à configurations en-ligne* : modélisent explicitement les configurations, mais spécifient les interconnexions des composants avec n'importe quel protocole d'interaction « en-ligne »,
- *Langages à configurations explicites* : modélisent les composants et les connecteurs séparément des configurations.

[3] *Off-The-Shelf Connectors*

Suivant la définition introduite dans le premier chapitre, les éléments de la première catégorie (langages à configurations implicites) ne sont pas des ADLs. Quoiqu'ils soient utilisés pour modéliser certains aspects de l'architecture. Nous citons, comme exemple dans cette catégorie de langage LILEANNA [Tracz 1993] où les informations sur les interconnexions sont distribuées entre les clauses « *with* » des packages individuels, « *view* » de l'assemblage des packages et « *make* » de leur composition.

L'aspect conceptuel de l'architecture et du traitement explicite des connecteurs font la différence entre les ADLs, les MILs[4] [Prieto-Diaz et Neighbors 1989], les langages de programmation orientés objets (exemple de Java), et les langages de notations orienté objets (exemple d'UML). Principalement les MILs décrivent les relations « *uses* » entre modules d'implémentation d'un système et supportent uniquement un seul type de connexion. Les langages de programmation décrivent l'implémentation des systèmes où l'architecture est pratiquement implicite dans la définition des sous-programmes et leurs appels.

2.3.1 Caractéristiques des configurations

Les configurations architecturales explicites facilitent la communication entre les différents intervenants du système qui ont des niveaux d'expertises variés par rapport au problème en question. Ceci est rendu possible par l'abstraction des détails de chaque composant et de chaque connecteur pour représenter la structure du système à un niveau élevé d'abstraction.

Parmi les caractéristiques importantes au niveau des configurations architecturales et que nous allons étudier dans les sections suivantes, figurent ceux qui sont liés à la qualité de description de configuration comme la composition, le raffinement et la traçabilité, ceux qui sont liés à la qualité du système décrit comme le passage à l'échelle et l'évolution et enfin ceux qui sont liés aux propriétés du système à décrire comme les contraintes.

2.3.1.1 Composition

La composition (ou composition hiérarchique) est un mécanisme qui permet aux architectures de décrire le système logiciel à différents niveaux de détail. La structure et le comportement complexe doivent être explicitement représentés ou doivent être abstraits sous forme d'un composant ou d'un connecteur. D'autres situations peuvent apparaître dans lesquelles une architecture entière devient un seul composant dans une autre architecture plus large. Un tel mécanisme d'abstraction devrait être fourni en tant que capacité de modélisation des ADLs. La plupart des ADLs fournissent des mécanismes explicites pour prendre en charge la composition hiérarchique des composants, où la spécification syntaxique des composants composites ressemble généralement à celle des configurations. La composition hiérarchique est utilisée dans les approches descendantes par raffinements successifs.

[4] *MILs : Module Interconnection Languages*

2.3.1.2 Raffinement et Traçabilité

Le raffinement et la traçabilité fournissent un cadre sûr pour la composition (une description de plus en plus détaillée). Ils sont nécessaires pour combler l'écart sémantique entre les modèles de haut niveau (diagrammes d'architecture en boites et lignes) informels et les modèles de bas niveau (langages de programmation et code) considérés comme des niveaux trop bas pour les activités de conception d'application [Moriconi *et al.* 1995].

Le critère fort du raffinement est la préservation qui assure que toutes les décisions faites à un niveau seront maintenues dans tous les niveaux suivants et rejette l'introduction de nouvelles décisions durant les phases intermédiaires du raffinement.

Les ADLs existants offrent un support limité au mécanisme de raffinement et de traçabilité. Plusieurs ADLs permettent la génération directe de l'implémentation des systèmes à partir de leurs spécifications architecturales. Ces ADLs sont des exemples typiques de « *Implementation Constraining Languages* » dans lesquels un fichier source correspond à chaque élément architectural. Cette approche de raffinement d'architecture est à l'origine de plusieurs problèmes. Principalement, il y a une hypothèse stipulant que la relation entre les éléments d'une description architecturale et ceux de son système exécutable doit être un-à-un. Ceci est inutile voire non raisonnable, du moment que les architectures décrivent les systèmes à un niveau d'abstraction plus élevé que le niveau des modules de code source. De plus, il n'y a aucune garantie que les modules de code source implémentent correctement l'élément architectural souhaité. Enfin, même si les modules spécifiés actuellement implémentent correctement le comportement souhaité, cette approche ne fournit aucun moyen pour assurer que les modifications que nous pouvons opérer par la suite sur ces modules soient retracées au niveau architecture et vice-versa.

2.3.1.3 Passage à l'échelle

L'ultime détermination qu'un ADL supporte bien le passage à l'échelle est de permettre aux développeurs d'implémenter et/ou d'analyser de larges systèmes basés sur la description architecturale d'un ADL. Plusieurs ADLs ont été utilisés pour décrire des architectures d'applications de grandes tailles. Les exemples suivants illustrent nos propos :

- Wright a été utilisé pour modéliser et analyser l'infrastructure d'exécution (RTI[5]) du ministère de la défense américaine dont la spécification initiale était plus de 100 pages [Allen *et al.* 1998].
- C2 a été utilisé dans la spécification et l'implémentation de son environnement support, composé d'un nombre important de composants spécifiques (100.000 LOC[6]) et de composants sur étagères « *OTS*[7] *Components* » (plusieurs millions LOC) [Medvidovic *et al.* 1999].

[5] *RTI : Run Time Infrastructure*
[6] *LOC : Line of Code*
[7] *OTS : Off The Shelf*

- L'exemple représentatif de l'utilisation de l'ADL Rapide est le « *X/Open Distributed Transaction Processing Industry Standard* », dont la taille de sa documentation est plus de 400 pages [Luckham *et al.* 1995].
- xADL 2.0 a été utilisé dans la spécification et la simulation du système d'aviation AWACS (*U.S. Airborne Warning and Control System*), dont la taille est estimée à 10.000 lignes de code xADL 2.0 [Milligan 2000].

Les ADLs doivent donc supporter directement la spécification et le développement de systèmes à grande échelle susceptibles de grandir encore. Nous considérons l'impact du passage à l'échelle d'une architecture suivant deux grandes dimensions :

- *l'ajout d'éléments à l'intérieur d'une architecture,*
- *l'ajout d'éléments aux frontières de l'architecture.*

2.3.1.4 Evolution

L'évolution d'une configuration architecturale est nécessaire pour que cette dernière puisse prendre en compte de nouvelles fonctionnalités et faire évoluer l'application sous-jacente. Cette évolution peut être considérée à partir de deux points de vue différents. Le premier est caractérisé par la capacité d'ajout, de suppression, de retrait et de reconnexion progressive des composants et des connecteurs dans une configuration. Le deuxième point de vue de l'évolutivité est la tolérance et/ou le support de l'ADL pour les descriptions incomplètes d'architectures.

Les architectures incomplètes sont fréquentes lors de la conception, vu que certaines décisions sont reportées et d'autres ne sont pas encore pertinentes. Il serait donc avantageux pour un ADL de permettre des descriptions incomplètes d'architectures. Toutefois, la plupart des ADLs existants ainsi que les outils de supports associés ont été construits avec l'idée d'éviter ce genre de situations à l'avance.

Un autre aspect important de l'évolution est le support du concept de familles d'applications. Pour qu'un ADL supporte le concept de familles, il est nécessaire d'établir une séparation explicite entre les types et les instances de composants et de connecteurs. Chaque instanciation d'une architecture peut être considérée comme un membre de la même famille. Ceci reste dépendant de la notion de famille car elle ne permet pas à l'architecture elle-même d'être modifiée. En outre, la famille à laquelle une application appartient est implicite dans son architecture.

2.3.1.5 Contraintes

Les contraintes décrivent les dépendances dans la configuration, et s'ajoutent à celles des composants et connecteurs. Plusieurs contraintes globales sont dérivées ou dépendent directement des contraintes locales. Les contraintes sur les configurations valides doivent être exprimées comme des contraintes d'interaction entre les constituants (composants et connecteurs), qui sont, à leur tour, exprimés à travers leurs interfaces et protocoles; ainsi les performances d'un système décrit par une configuration, dépendent des performances individuelles de chaque élément architectural.

2.4 Les langages de description d'architectures

L'étude de Medvidovic et Taylor [Medvidovic et Taylor 2000] a révélé qu'il y eu une prolifération dans les ADLs disponibles, tandis qu'il y avait beaucoup de similitudes entre eux. Néanmoins, ces ADLs ont différentes vocations, ce qui suggère la possibilité d'utiliser plusieurs ADLs pour décrire l'architecture d'un même système. Nous présentons dans la Figure 2.1 une frise chronologique des différents ADLs qui ont été proposés, en prenant soin de mentionner leur origine. Cette rétrospective fait ressortir deux générations d'ADLs issus de trois écoles académiques.

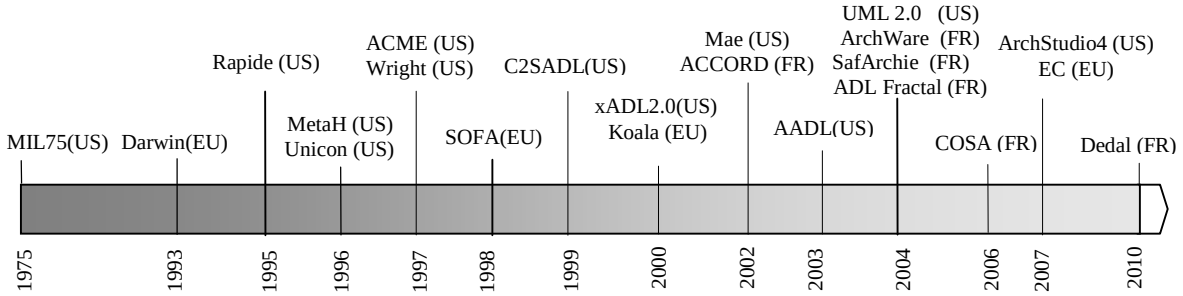


Figure 2.1- Frise chronologique des ADLs et leur origine.

2.4.1 Générations d'ADLs

Dans la dernière décennie, une douzaine d'ADLs différents sont apparus. Chacun d'eux vise à décrire une configuration architecturale, mais selon un angle de vue différent. En outre, chacun de ces ADLs possède des règles de syntaxe très spécifiques adaptées à la description de certaines propriétés de l'architecture. De nos jours, nous faisons référence à ces notations comme des ADLs de première génération. Certains des exemples les plus représentatifs de cette génération sont Darwin, Wright, Rapide et C2 [Medvidovic et Taylor 2000].

L'étude de la première génération d'ADL a montré que, bien que différents, ces ADLs partagent une même base conceptuelle qui détermine un ensemble de concepts et de préoccupations communes pour la description architecturale. Cela a naturellement conduit à l'idée qu'il serait possible de créer un ADL générique, qui combinerait les propriétés communes et les caractéristiques bénéfiques de la première génération d'ADL. Ces langages qui cherchent à prendre du recul sur les tentatives antérieures pour la spécification d'architectures sont considérés comme des ADLs de seconde génération. Les exemples les plus significatifs de la deuxième génération sont ACME [Garlan *et al.* 1997], xADL 2.0 **[**Dashofy *et al.* 2005**]** avec ArchStudio 4 comme environnement de modélisation et de méta-modélisation d'architectures [Dashofy *et al.* 2007]. En acceptant l'idée de considérer UML 2.0 comme un ADL à part entière, celui-ci s'inscrit dans cette seconde génération.

2.4.2 Écoles académiques des ADLs

Le paysage académique des ADLs se partage entre l'école américaine (US), l'école européenne (EU) et l'école française (FR). Historiquement, MIL75 [DeRemer et Kron 1975] - considéré comme l'ancêtre des ADLs – fût avancé par les américains, dont les travaux sont restés très influents jusqu'à aujourd'hui. La contribution de l'école américaine est issue essentiellement de la CMU (*Carnegie Mellon University*) et l'UCI (*University of California, Irvine*). Dans le même temps, UML 2.0 est un poids lourd du domaine qui est formalisé par OMG (*Object Managment Group*). De son coté l'école européenne s'est illustrée très tôt avec la contribution de l'*Imperial College* avec Darwin [Magee *et al.* 1995], puis plus tard avec Koala [Ommering *et al.* 2000], quand à SOFA [Plásil *et al.* 1998], c'est un ADL développé par la *Charles University*, située en République Tchèque et EC (Exogenous Connectors) une approche de composition de connecteurs développée par l'équipe de Kung-Kiu Lau à l'*Université de Manchester* [Lau *et al.* 2007]. L'école française a commencé à émerger ces dernières années, avec le projet ACCORD [ACCORD 2002], ADL Fractal [Bruneton 2004], SafArchi [Barais et Duchien 2004], ArchWare [Oquendo 2004], COSA [Smeda 2006] un ADL proposé par notre équipe et récemment Dedal un ADL proposé par *LGI2P / Ecole des Mines d'Alès, Nîmes* [Zhang *et al.* 2010]. En dehors du cercle académique ; on notera tout de même la participation des industriels aux ADLs MetaH (*Honeywell*) et Koala (*Philips*).

2.4.3 École américaine (US)

2.4.3.1 Wright

Wright [Allen, 1997] a été une des premières approches à permettre la description du comportement de composants. Il se centre sur la spécification de l'architecture et de ses éléments. Il n'y a pas de générateur de code, ni de plate-forme permettant de simuler l'application comme pour Rapide [Luckham *et al.* 1995]. Les trois notions de base d'un ADL à savoir le composant, le connecteur et la configuration sont présentes dans Wright.

La spécification de Wright permet de décrire à la fois les composants et les connecteurs composites : la fonction d'un composant composite et la glu d'un connecteur composite sont représentées par une description architecturale plutôt qu'en CSP.

Le connecteur Wright

Un connecteur Wright représente une interaction entre une collection de composants. Il spécifie le type d'un protocole interaction de manière explicite et abstraite. Ce type peut être réutilisé dans différentes architectures. Par exemple, un protocole de validation à deux phases (*two-phase-commit*) peut être un connecteur.

Dans l'ADL Wright, un connecteur contient deux parties importantes qui sont un ensemble de rôles et la glu. Chaque rôle indique comment se comporte un composant qui participe à l'interaction. Le comportement du rôle est décrit par une spécification CSP. La glu décrit comment les participants (les rôles) interagissent entre eux pour former une

interaction. Par exemple, la glu d'un connecteur « appel de procédure » indiquera que l'appelant doit initialiser l'appel et que l'appelé doit envoyer une réponse en retour. Enfin, Wright supporte l'évolution des connecteurs via le paramétrage ; par exemple, le même connecteur peut être instancié avec une glu différente.

La configuration Wright

La configuration permet de décrire l'architecture d'un système en regroupant des instances de composants et des instances de connecteurs. La description d'une configuration est composée de trois parties qui sont la déclaration des types des composants et des connecteurs utilisés dans l'architecture, la déclaration des instances de composants et de connecteurs, les descriptions des liens entre les instances de composants et des connecteurs.

Wright supporte la composition hiérarchique. Ainsi, un composant peut être composé d'un ensemble de composants. Il en va de même pour un connecteur. Lorsqu'un composant représente un sous-ensemble de l'architecture, ce sous-ensemble est décrit sous forme de configuration dans la partie calcul du composant.

D'autre part, plusieurs ADLs fournissent des facilités pour spécifier des contraintes globales et arbitraires. Wright permet la spécification des invariants structurels correspondants à de différents styles architecturaux.

Un exemple de contrainte de style dans l'ADL Wright est donné dans la Figure 2.2. Cette dernière montre que la contrainte sur le style, spécifie que tous les connecteurs sont de type *Pipe* et tous les ports de composants sont de type *DataInput* ou *DataOutput*. MetaH spécifie des contraintes explicites sur les configurations, de la même manière que les composants, en utilisant des propriétés non-fonctionnelles.

```
Style Pipe-Filter
   ……
    Constraints
       ∀ c : connector •  Type(c) = Pipe
     ∧ ∀ c : Component; p : Port | p ∈  Ports(c) •
              Type(p)= DataInput ∨ Type(p)= DataOutput
```

Figure 2.2- Déclaration du style Pipe-Filter dans l'ADL Wright.

2.4.3.2 ACME

ACME est un langage de description d'architecture établi par la communauté scientifique dans le domaine des architectures logicielles. Il a pour buts principaux de fournir un langage pivot qui prend en compte les caractéristiques communes de l'ensemble des ADLs, qui soit compatible avec leurs terminologies et qui propose un langage permettant d'intégrer facilement de nouveaux ADLs.

ACME propose trois concepts de base pour définir une architecture à savoir le *composant*, le *connecteur* et le *système* qui représente la configuration d'une application. Les

points d'interface des connecteurs dans ACME sont appelés des « *rôles* », qui sont nommés et typés. Dans l'ADL ACME les types de connecteurs sont spécifiés par des protocoles d'interactions. ACME fournit des facilités de paramétrage qui permettent une spécification flexible des signatures de connecteurs et des contraintes sur la sémantique des connecteurs. Plusieurs ADLs emploient les mêmes mécanismes utilisés dans l'évolution des composants pour faire évoluer les connecteurs ; ACME définit le sous-typage structurel des connecteurs comme mécanisme pour leur évolution. La Figure 2.3 illustre la spécification de l'interface du connecteur *http* dans une simple application web écrite en ACME.

```
    System web_application_1 = {
        Component web_client = {Port send-http-request;};
        Component web_server = {Port receive-http-request;};
        Connector http = {Roles {sender, receiver}};
        Attachments { web_client.send-http-request to http.sender;
                      web_receive-http-request to http.receiver;
        }
    }
```

Figure 2.3 - Spécification ACME d'une simple application web.

La Figure 2.4 présente la même description que la Figure 2.3 (application web), où les éléments architecturaux sont annotés avec des propriétés supplémentaires. Ces propriétés augmentent la précision de la spécification et capturent, par exemple, l'origine précise de la version des différents composants.

```
System web_application_2 = {
      Component web_client = {
            Port send-http-request;
      Property vendor : Microsoft;
            Property product: Internet Explorer;
            Property version: 7.0sp1;
            Property supported-http-versions: 1.0, 1.1;
      };
      Component web_server = {
            Port receive-http-request;
            Property vendor : Apache;
            Property product: httpd;
            Property version: 2.0.48;
            Property supported-http-versions: 1.0, 1.1;
      };
      Connector http = {
            Roles {sender, receiver}
            Property http-version: 1.1;
      };
      Attachments {
            web_client.send-http-request to http.sender;
            web_server.receive-http-request to http.receiver;
      }
}
```

Figure 2.4 - Description annotée avec des propriétés pour l'application Web dans ACME.

ACME figure parmi les quelques ADLs qui spécifient explicitement le concept de famille d'architectures, comme une entité de première classe du langage, et supporte leurs opérations d'évolution. Les types de composants et de connecteurs déclarés dans une famille offrent un vocabulaire bien précis de conception pour tous les systèmes déclarés comme étant membres de cette famille. L'exemple donné dans la Figure 2.5 présente la déclaration ACME d'une famille d'architecture (*fam*) et une sous-famille (*sub_fam*). Où (*sub_fam*) est l'évolution de (*fam*) par l'ajout du composant (*comp3*) et d'une propriété à un port du composant (*comp1*).

```
Familly fam = {
    Component Type comp1 = {Port p1 ;}
    Component Type comp2 = {Port p2 ;}
    Connector Type conn1 = {Roles (r1, r2);}
}
Familly sub_fam extends fam with {
     Component Type sub_comp1 = comp1 with {
         Port p1 = {Property attach: int <<default = 1>> ;}
    }
    Component Type comp3 = {. . .}
}
```

Figure 2.5- Déclaration d'une famille d'architecture et son évolution en ACME.

2.4.3.3 C2 (C2SADL)[8]

C2 est un langage de description d'architecture conçu par l'université UCI en Californie [Medvidovic *et al.* 1999]. Il fut conçu, à l'origine pour la conception des interfaces graphiques et s'étend aujourd'hui à la conception d'autres applications. Il possède trois abstractions principales qui sont le composant, le connecteur et la configuration d'une architecture. L'idée de cet ADL est de définir une application sous forme de réseau de composants s'exécutant de manière concurrente, liés par des connecteurs et communiquant de manière asynchrone par envoi de messages.

Connecteur C2

Dans C2 les types de connecteurs sont spécifiés par des protocoles d'interactions. Les interfaces de connecteurs dans C2 sont génériques : les connecteurs sont indifférents aux types de données qu'ils manipulent ; leur tâche principale est d'arbitrer et de coordonner la communication entre composants. En plus, les connecteurs de C2 peuvent supporter un nombre arbitraire de composants en interactions. En C2, cette fonctionnalité est appelée *contexte réflexive*: l'interface d'un connecteur est déterminée par (potentiellement dynamique) les interfaces des composants qui communiquent à travers elle, comme le représente la Figure 2.6.

[8] "C2SADL" est le nom complet de l'ADL SADL suivant le style d'architecture C2

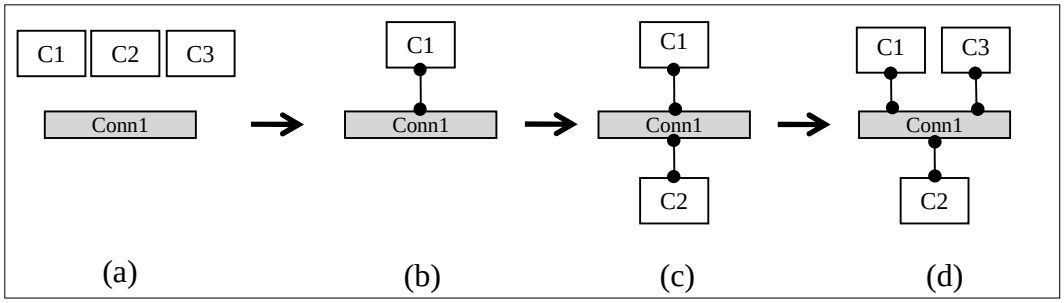


Figure 2.6 - Connecteur à contexte réflexive de C2 : (a) L'architecte sélectionne à partir de la bibliothèque un ensemble de composants et un connecteur ; (b-d) au fur et à mesure que les composants sont attachés au connecteur pour former l'architecture, le connecteur crée de nouveaux ports de communication pour supporter la communication inter-composants.

Les connecteurs dans C2 sont par nature évolutifs en raison de leurs interfaces contexte-réflexives. Ils peuvent aussi évoluer en modifiant leur politique de filtrage. C2 n'offre aucune prise en charge des propriétés non-fonctionnelles.

Configuration C2

C2 permet de définir la configuration d'une architecture en spécifiant :

- La structure de l'application, i.e., les composants utilisés et les connecteurs qui assurent les interactions entre ces composants,
- La dynamique de l'application, i.e., les changements de l'architecture au cours de l'exécution comme, par exemple l'ajout ou le retrait de composants,
- La spécification de l'application est composée de deux parties : (a) la spécification de l'architecture logicielle, (b) la spécification de l'implémentation.

(a)

```
architecture ::=
    architecture architecture_name is
    conceptual_components conceptual_component_list
    [connector connector_list]
    [architectural_topology topology]
end architecture_name;
```

(b)

```
system ::=
system system_name is
    architecture architecture_name with
    component_instance_list
end system_name;
```

La partie *topology* décrit l'assemblage statique des connecteurs avec les composants. Elle offre aussi la possibilité d'exprimer une partie de la dynamique d'une architecture grâce aux contraintes de connexion (clause *'where'*) dont les valeurs déterminent si une instance de composant doit faire partie de l'architecture.

Dans la Figure 2.7 nous présentons un exemple d'une configuration en C2 dans lequel nous illustrons deux cas d'expansion d'une architecture. (a) ajout d'un nouveau composant *comp5* à l'intérieur de l'architecture. (b) ajout d'un nouveau composant *comp6* et d'un nouveau connecteur *conn3* au frontière (expansion de l'architecture vers l'extérieur).

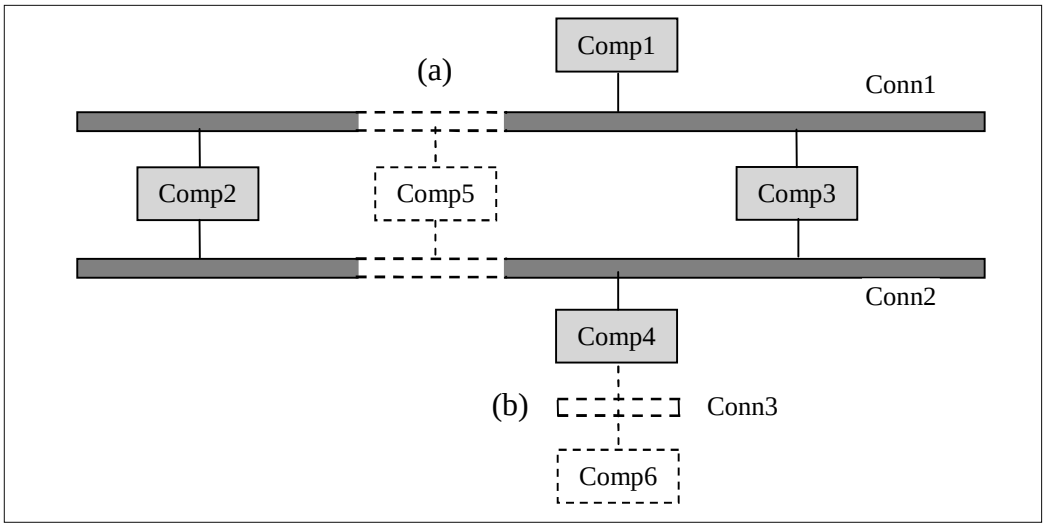


Figure 2.7 Passage à l'échelle d'une architecture décrite en notation graphique de C2.

Pour mettre en œuvre le deuxième cas, les ADLs peuvent, au minimum, utiliser le mécanisme de composition discuté dans la section 2.3.1.1, où l'architecture d'origine est traitée comme un seul élément composite, attaché ensuite à de nouveaux composants et connecteurs. L'évaluation objective portant sur la capacité d'un ADL à mettre en œuvre le premier cas est difficile, mais certaines heuristiques peuvent aider à le faire.

Généralement, il est plus facile d'étendre les architectures décrites dans les ADLs supportant les configurations explicites que dans les ADLs supportant les configurations en-ligne. Les connecteurs dans les configurations en-ligne sont décrits seulement en termes de composants qu'ils relient et l'ajout de nouveaux composants nécessite des modifications au niveau des instances existantes de connecteurs. Par contre, les ADLs qui permettent à un nombre variable de composants à être attachés à un seul connecteur sont mieux adaptés au passage à l'échelle que ceux qui spécifient qu'un connecteur peut servir uniquement un nombre fixe de composants.

2.4.3.4 xADL2.0

xADL2.0 fournit un ensemble de schémas XML qui définissent un ensemble initial de concepts architecturaux. Cet ensemble de schémas lui-même est *modulaire* et *extensible*.

Chaque concept du langage (composant, connecteur, interface et configuration) est défini dans un schéma séparé et chaque schéma peut servir de base pour d'autres extensions.

xADL2.0 possède une inhérente prise en charge de l'évolution, non seulement de toute l'architecture, mais aussi des éléments qui composent une architecture. En plus de garder une trace précise des changements qui sont apportés à une architecture au fil du temps, cette unique capacité de xADL2.0 permet de modéliser des architectures multi-version (*versioning*). Ainsi, nous pouvons conclure que l'évolution des connecteurs dans xADL2.0 est inhérente via le mécanisme de sous-typage qui permet la spécialisation des schémas de connecteurs [Khare *et al.* 2001] **[**Dashofy *et al.* 2005**]**.

2.4.3.5 AADL

AADL (Architecture Analysis & Design Language) [Feiler *et al.* 2003] est un langage de description d'architectures avioniques, sous l'autorité du SAE (Society of Automotive Engineers), organisme de standardisation mondiale dans le domaine du transport. Son étude est intéressante pour au moins deux raisons. Ce langage issu d'un organisme de normalisation rencontre un très bon accueil auprès des industriels du domaine.

AADL vise à répondre aux besoins spéciaux des systèmes embarqués temps réel tels que les systèmes avioniques. En particulier, il peut décrire les mécanismes standards de flots de contrôles et de données utilisés dans de tels systèmes, et des aspects non fonctionnels importants tels que les exigences temporelles, les fautes et erreurs, le partitionnement temporel et spatial et les propriétés de sûreté et de certification.

AADL est à l'origine défini par différents partenaires du domaine de l'avionique, mais il s'adresse dans les faits à tous les systèmes embarqués temps réel, et pas uniquement aux systèmes avioniques. AADL est d'abord un langage textuel avec une annexe standard qui définit la notation graphique associée et une autre annexe définit un profil UML pour AADL.

Dans AADL les composants peuvent avoir des sous-composants décrits dans leurs implémentations en utilisant des règles de composition. AADL utilise deux concepts pour désigner une configuration. Les *systèmes* qui permettent de structurer l'architecture et contiennent des composants qui peuvent être manipulés de façon indépendantes, et les *composants abstraits* qui sont de simples conteneurs sans aucune sémantique et permettent aussi de structurer l'architecture.

Les connecteurs AADL

Les interactions entre les composants sont matérialisées par la définition de connexions entre leurs ports. Une connexion permet de relier deux ports. Les ports peuvent être déclarés en entrée (*in*), sortie (*out*), ou entrée-sortie (*in out*). Une vérification est faite qu'il y a bien conformité de type et de sens entre les ports connectés. Les connecteurs n'ont pas de comportement associé, mais il est possible de leur ajouter des propriétés sur le délai de transmission des données.

La configuration AADL

De par la nature même des composants, le modèle d'AADL est hiérarchique. Par exemple, un processus peut contenir des processus légers. En outre, la notion de *système* qui peut lui même être composé de sous-systèmes permet de décomposer le système sur une infinité de niveaux. La Figure 2.8 montre un système contenant deux processus, eux-mêmes contenant chacun un processus léger. Une succession de ports et de connexions établit une liaison entre les deux processus légers.

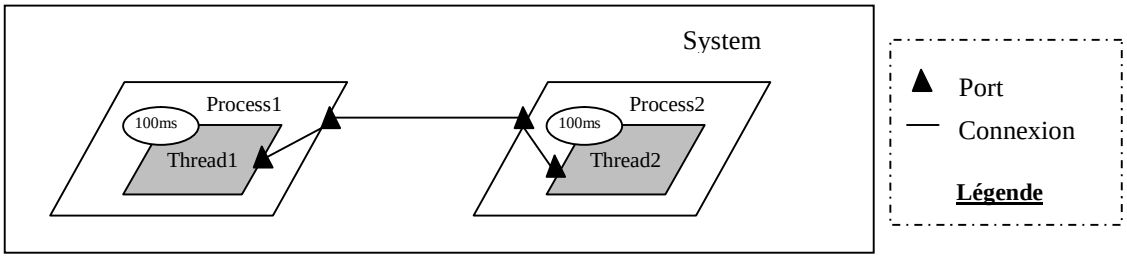


Figure 2.8 – Exemple simple d'une architecture AADL.

2.4.3.6 UML 2.0

Après le succès de la première version d'UML [OMG 2003b], la deuxième version notée UML 2.0 [OMG, 2003c] introduit la notion de diagramme d'architecture, aussi appelée diagramme de structure composite. UML propose une notation unique pour couvrir toutes les phases du développement logiciel. Avec l'ajout de ce diagramme, UML veut combler une des lacunes de la première version en permettant la spécification de l'architecture d'une application.

UML 2.0 a été considérablement révisé afin de mieux capturer l'aspect structurel de l'architecture, en utilisant des éléments similaires à ceux trouvés dans d'autres ADLs tel que Darwin. Les connecteurs explicites ne sont pas encore définis dans UML 2.0, quoiqu'en cas de besoin les composants peuvent être adaptés pour décrire les connecteurs [Medvidovic *et al.* 2007].

Le connecteur UML 2.0

Comme pour les ADLs, les interactions entre les composants sont décrites par des connecteurs. Un connecteur est une entité qui relie des ports ou des interfaces de composant. Dans un diagramme d'architecture, une application ou un composant est constitué de l'assemblage d'autres composants par l'intermédiaire de ports interconnectés. Il existe deux types de connecteur :

- le connecteur d'assemblage qui permet d'assembler deux instances de composant en connectant un port fourni d'un composant au port requis de l'autre composant,
- le connecteur de délégation qui permet de connecter un port externe au port d'un sous-composant interne. L'invocation d'un service sur le port externe sera transmise au port interne auquel il est connecté.

Plusieurs connecteurs peuvent être attachés au même port ou à la même interface.

La configuration UML 2.0

L'introduction du concept de partie (*Part*) dans UML 2.0 rend possible la description de la partie interne d'une classe. Une partie est donc une propriété de la classe, c'est-à-dire elle a le même cycle de vie que l'objet de la classe à laquelle elle appartient. Les parties sont un ensemble d'instances d'autres classes. Comme le composant en UML 2.0 étend la notion de classe, sa structure interne est elle aussi décrite à l'aide de parties qui peuvent être respectivement des instances de classe ou des instances de composant. Si ces parties sont des instances de composant, il est possible de définir l'assemblage entre les ports de ces instances par l'intermédiaire de connecteurs.

Une partie représente une ou plusieurs instances d'une classe ou d'un composant grâce à des contraintes de multiplicité. Sans contrainte de multiplicité, l'objet est construit dès que la partie est construite. La multiplicité permet d'indiquer le nombre minimum et maximum d'instances au sein de la part mais aussi le nombre d'instances créées automatiquement lors de la création de la classe ou du composant contenant le part.

UML 2.0 permet donc, comme dans la plupart des langages de description d'architecture, de définir son architecture de manière hiérarchique. En effet, la notion de part permet de définir la réalisation d'un type de composant à partir d'un assemblage de composants.

2.4.4 École européenne (EU)

2.4.4.1 Darwin

Darwin [Magee *et al.* 1995] a été développé au "*Distributed Software Engineering Group*" de l'Imperial Collège de Londres. Il propose un modèle de composant pour la construction d'applications distribuées. Sa particularité est de permettre la spécification d'une partie de la dynamique d'une application en termes de schéma de création de composants logiciels avant, pendant ou après l'exécution d'une application.

Le modèle de Darwin reprend les trois concepts de base des modèles à composants à savoir, le composant, le connecteur et la configuration. Cependant le connecteur n'a pas une sémantique de processus. Wright n'offre aucune prise en charge des propriétés non-fonctionnelles.

Le connecteur Darwin

L'interaction entre les composants se fait par l'intermédiaire des services offerts et requis par le composant. Le concept de connecteur n'est pas explicitement présent dans Darwin. En effet, chaque interaction est représentée par un lien entre un service fourni et un service requis de composants différents. Les services n'ont aucune connotation fonctionnelle, ils désignent seulement le type d'objet de communication utilisé et autorisé à utiliser une fonction du composant. Ces types d'objets sont définis par le support d'exécution répartie appelé *Regis* et sont limités.

Dans une description d'architecture exprimée avec Darwin, la section « *port* » caractérise le type d'objet de communication, la section « *signature* » décrit la signature de la fonction du composant. Darwin par défaut effectue un simple test d'équivalence des noms et des types avant la connexion, vérifiant que le service requis est compatible avec le service fourni.

La configuration Darwin

Afin d'offrir une hiérarchie au sein d'une application, Darwin contient deux sortes de composant :

- les composants *primitifs* sont avant tout des entités d'encapsulation de fonctions et de données d'un module logiciel. Ils sont définis par leur nom et leur interface qui déclarent les services requis et fournis par le composant métier,
- le composant *composite* contient des composants primitifs et des composants composites. C'est donc une unité de configuration. Il décrit les interconnexions entre les composants. Ainsi, la structure finale d'une application est représentée par un composant composite.

L'ADL Darwin n'a pas de construction explicite pour la modélisation des configurations. Il les modélise comme étant des composants composites. Un exemple d'illustration du support de Darwin pour les composants composites est donné dans la Figure 2.9.

(a)

```
Component Composite {
    provide provService;
    require reqService;
    instances
        C1 : CompType1;
        C2 : CompType2;
    Bind
      provService –- C1.pService;
      C2.rService –- reqService;
}
```

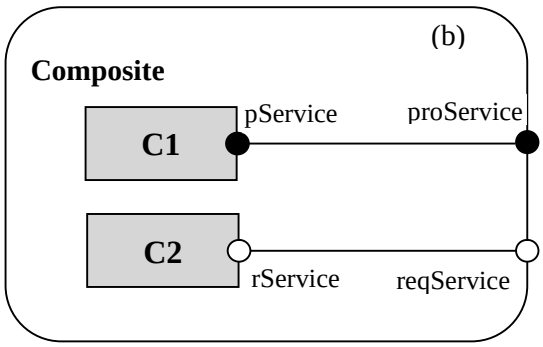


Figure 2.9- (a) : Description syntaxique d'un composant composite avec l'ADL Darwin,
(b) : Description graphique du composite.

2.4.4.2 SOFA

SOFA (*SOFtware Appliance*) [Plásil *et al.* 1998] est un modèle de composant développé par l'université Charles (Prague). Son objectif est la construction d'application à partir d'une composition de composants. SOFA propose les éléments caractéristiques des langages de description d'architecture, à savoir le composant, le connecteur, et la configuration. SOFA est sans aucun doute l'approche la plus à cheval entre un langage de description d'architecture ou un modèle de composant académique.

Le connecteur SOFA

Les connecteurs dans SOFA sont des éléments de premier ordre au même titre que les composants. Ils n'ont pas fondamentalement de différence avec les composants. Ils sont définis à l'aide de *connector-frame* pour la vision boîte noire et *connector-architecture* pour la vision boîte grise. L'objectif des connecteurs de SOFA est de permettre de capturer uniquement les logiques métiers au sein des composants pendant que les connecteurs prennent en charge la sémantique de l'interaction et les problèmes de déploiement.

Les *connector-frames* sont définis à partir d'un ensemble de rôles. Un rôle est une interface générique pouvant être liée à une interface de composant. Le *connector-architecture* décrit la partie interne du connecteur. Comme pour le composant, cette partie peut être déclarée primitive directement implantée ou elle peut être déclarée composée, elle contient alors un ensemble de composants et de connecteurs.

SOFA fournit trois types de connecteur prédéfinis : *CSProcCall* avec une sémantique d'appel de procédure, *EventDelivery* avec une sémantique d'envoi de message et *DataStream* pour une sémantique d'échange de flux. Il fournit aussi un type de connecteur utilisateur qui permet de définir sa propre sémantique. Cette dernière sera alors réalisée à l'aide d'un assemblage de composants et de connecteurs. De manière implicite, si aucun connecteur n'est spécifié, un connecteur possédant une sémantique d'appel de procédure est introduit.

La configuration SOFA

Il y a quatre moyens de venir s'accrocher à une interface de composant. La première appelée le *binding* permet de créer une liaison entre une interface requise et une interface fournie de deux composants de même niveau. La deuxième appelée *delegating* lie une interface fournie du composant englobant à une interface fournie d'un de ses sous-composants. La troisième nommée *subsuming* permet de lier un port requis d'un sous-composant à un port requis du composant englobant. Enfin, la dernière nommée *exempting* permet de déclarer qu'une interface d'un composant n'est pas liée. Les trois premiers types d'accroche sont réalisés par des connecteurs.

SOFA comme Fractal propose une approche statique de la description d'une architecture logicielle. Il ne propose pas un modèle d'invariants du système permettant de définir un cadre pour la dynamique de l'architecture. Ainsi, une architecture représente un assemblage de composants à un instant "t". Dès lors, même si la mise en œuvre du modèle permet le remplacement à l'exécution de composants, la prise en compte de la dynamique au niveau du modèle est assez restreinte.

2.4.4.3 Koala

Koala [Ommering *et al.* 2000] est inspiré de Darwin. Les composants sont définis dans un ADL. Chaque composant possède un ensemble d'interfaces fournies et requises. Les interfaces de composants sont définies dans un IDL.

La configuration Koala

Une configuration dans Koala est un ensemble de composants connectés via leurs interfaces requises et fournies. Plusieurs interfaces requises peuvent être connectées à une seule interface fournie. Les interactions complexes sont définies dans des modules (i.e., des composants sans interfaces). Ces derniers implémentent l'ensemble des fonctions de toutes les interfaces impliquées dans une interaction complexe et jouent le rôle de connecteurs entre composants. Ils sont déclarés à l'intérieur des composants.

La Figure 2.10 illustre la notation graphique utilisée pour définir une configuration Koala. Par contre, la Figure 2.11 représente la description de la même configuration dans l'ADL Koala.

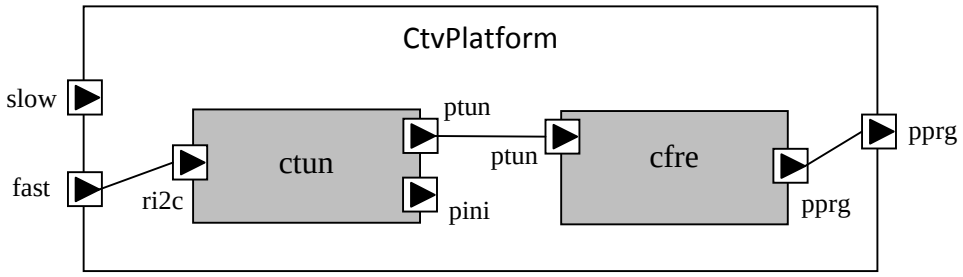


Figure 2.10 – Connexion d'interfaces des composants Koala.

```
Interface Ttuner {
      Void setFrequency (int f);
      Int GetFrequency (void);
}
Component CTunerDriver {
       Provides ITuner ptun; IInit pini;
       Requires II2c  ri2c;
}
Component CFrontEnd {
       Provides IProgram pprg;
       Requires ITuner ptun;
}
Component  CtvPlatform {
      Provides  IProgram pprg;
      Requires  II2c slow,fast;
      Contains
             Component CFrontEnd cfre;
             Component CTunerDriver ctun;
      Connects
            pprg = cfre.pprg;
             cfre.rtun = ctun.ptun;
             ctun.ri2c = fast;
}
```

Figure 2.11 - Spécification Koala d'une simple application de montage TV.

2.4.5 École Française (FR)

2.4.5.1 ADL Fractal

Malgré l'adoption des modèles à composants industriels comme EJB [DeMichiel 2003] ou CCM [OMG 2002b], la nécessité d'avoir un modèle plus léger et plus proche des concepts des langages de programmation s'est fait ressentir. Le modèle de composant Fractal [Bruneton et *al.* 2004] réalisé dans le cadre du consortium *ObjectWeb* par France Telecom R&D et par l'INRIA cherche à combler ce besoin. Fractal vise à autoriser une définition, une configuration et une reconfiguration dynamique d'une architecture à base de composants, ainsi qu'une séparation claire des préoccupations fonctionnelles et non-fonctionnelles.

En amont du modèle de composant Fractal [Bruneton *et al.* 2003], il est possible de décrire l'architecture d'une application à base de composants à l'aide de l'ADL Fractal. Basé sur XML, ce langage définit une syntaxe abstraite indépendante de tout langage de programmation pour la description de l'architecture en termes de composants, d'interfaces, de liaisons et d'attributs.

L'ADL Fractal est un ADL extensible. Il permet d'intégrer facilement de nouveaux concepts au niveau de la description de l'architecture en modifiant la grammaire du langage. Cette grammaire est en effet construite à partir d'un ensemble de modules prenant en charge chacun une caractéristique du modèle de composant comme les interfaces, les liaisons ou les attributs.

L'ADL Fractal permet de définir l'architecture structurelle d'un système comme étant un ensemble de composants. Ces deniers, sont décrits en termes de *contenu*, de *contrôleurs* et d'*interfaces* ainsi que leurs *liaisons*. Le contenu d'un composant est construit à partir d'un nombre fini de composants, appelés sous-composants. Le modèle de composant Fractal est récursif grâce à cette notion de sous-composant. La récursivité s'arrête au niveau d'un composant primitif. Le contrôleur d'un composant a la capacité d'intercepter les appels entrants et sortants d'un composant, de donner une représentation interne du contenu du composant ou encore de suspendre ou d'arrêter l'activité d'un sous composant. Les interfaces dans Fractal peuvent être de deux types : *client* ou *serveur*. Une interface serveur peut recevoir des invocations de services alors qu'une interface client en émet. Une liaison Fractal peut être établie entre ces deux interfaces. Ainsi, l'ADL Fractal est modulaire et extensible offrant la possibilité d'être étendu afin de spécifier, par exemple, des contraintes de déploiement ou de comportement.

La Figure 2.12 fournit une représentation graphique d'un composant Fractal *CompX* fournissant une interface *iX* et requérant une interface *iY*. Dans cet exemple, l'interface *iY* est fournie par le composant *CompY*. D'un point de vue fonctionnel, le composant *CompX* fournit *iX* si, et seulement si, un autre composant lui fournit *iY*.

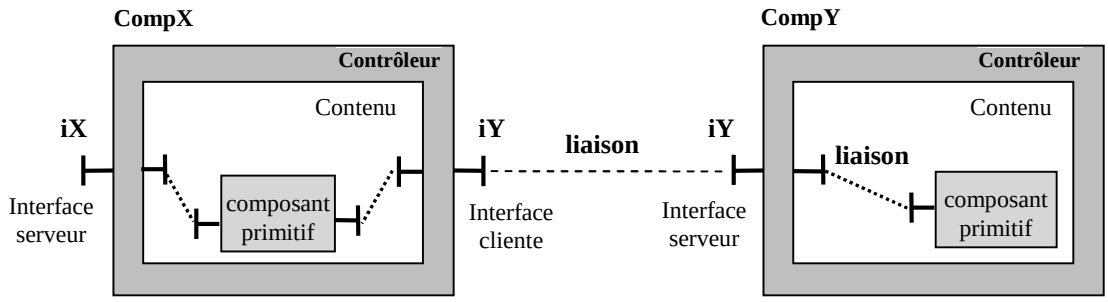


Figure 2.12- Représentation graphique de composition de composants avec l'ADL Fractal.

Le connecteur Fractal

Comme Darwin, Fractal ne possède pas de notion de connecteur explicite avec une sémantique de processus. Cependant, il utilise la notion de liaison pour spécifier les interactions entre composants. Une liaison Fractal est définie comme un lien orienté entre une interface cliente et une interface serveur. Ce lien permet aux composants d'interagir. Le modèle étant fortement typé, le type d'une interface serveur doit être obligatoirement du type ou du sous-type de l'interface cliente à laquelle elle est reliée. La liaison est assimilable à un connecteur implicite, elle ne possède pas de comportement propre.

La membrane (ensemble de contrôleurs) possède à la fois des interfaces externes, qui sont accessibles de l'extérieur du composant, et des interfaces internes, accessibles seulement par son contenu. Dans le modèle Fractal, une interface interne ne peut exister que symétriquement à une interface externe. Ce mécanisme d'interface interne sert principalement à permettre les traversées de membranes des composites en conservant la sémantique de la liaison. Ainsi la liaison permet de définir à la fois les interactions entre deux composants mais permet aussi de définir les liens de délégation entre les interfaces d'un composite et les composants de son contenu.

La configuration Fractal

Fractal est un modèle hiérarchique, un composant peut posséder au sein de son contenu d'autres composants. Il est identifié dès lors comme un composite. Le contenu d'un composite définit une configuration pour la mise en œuvre des services définis au niveau de ses interfaces métiers. Enfin, et c'est une spécificité du modèle Fractal, un composant peut appartenir au contenu de deux composites qui ne sont pas imbriqués l'un dans l'autre. Il est alors dit partagé [Bruneton *et al.* 2002].

2.4.5.2 Projet Accord

Dans le cadre du projet Accord l'équipe de Beugnard à ENST-Bretagne [Matougui et Beugnard 2005] [ACCORD 2002] définit un connecteur comme une entité abstraite qui doit assurer une propriété de communication. Il s'agit d'une potentielle qui se concrétise au fil du temps, en avançant dans le processus de développement logiciel. Un connecteur n'offre pas de services explicites qui seront utilisés par d'autres composants. Il propose

d'assurer une propriété de communication sans la réifier comme des services. Les blocs de construction suivant sont ajoutés au connecteur pour lui permettre d'assurer ces fonctionnalités :

Propriété : le connecteur met en œuvre la sémantique de communication ou coordination entre les composants. La *propriété* représente la fonctionnalité du connecteur qui doit être indépendante de la fonctionnalité des composants.

Prise : Le connecteur interagit avec le monde extérieur pour lui assurer sa propriété via des points d'attaches appelés *prises*. Celles-ci correspondent aux extrémités des canaux auxquels nous attachons une importance particulière. Les prises sont des interfaces puisqu'elles permettent l'échange d'information entre le connecteur et les autres composants.

Protocole : les prises du connecteur sont destinées à communiquer entre elles pour pouvoir assurer la propriété de communication ou de coordination aux composants qui y sont attachés. Ces prises communiquent à travers un *protocole*. Le protocole constitue un ensemble de règles qui régissent le fonctionnement du connecteur.

Générateur : le connecteur est réalisé comme un *générateur* de code. Le générateur utilise la plateforme sous-jacente et choisit une solution pour la réalisation de la propriété du connecteur sur cette plateforme. Il permet de garder l'abstraction de communication du connecteur intacte jusqu'au déploiement et de ne pas la perdre en la réalisant dans le code des composants interagissant. Les prises du générateur restent génériques, elles représentent des emplacements à combler avec les interfaces de la connexion. Lorsque ces interfaces sont passées en paramètres au générateur, ce dernier génère le *composant de liaison* qui représente une entité entièrement concrétisée. Le connecteur est transformé en un composant de liaison avec des interfaces bien définies en adoptant les interfaces de la connexion et en effectuant la communication effectives au-dessus de la plateforme choisie. La Figure 2.13 illustre les différents blocs de construction d'un connecteur.

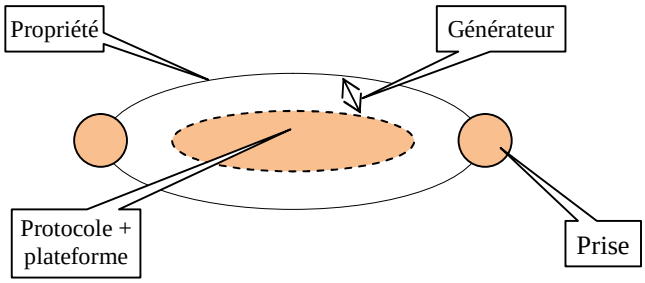


Figure 2.13- Structure d'un connecteur dans le projet Accord.

L'équipe du projet ACCORD n'aborde pas dans leurs travaux le concept de configuration et description hiérarchique des architectures logicielles.

2.4.5.3 SafArchie

SafArchie (*Safe Architecture*) [Barais et Duchien 2004] s'appuie sur une description étendue, à la fois structurelle et comportementale, des interfaces des composants afin de permettre une analyse de l'assemblage. Il est basé sur deux modèles de composant abstraits permettant de spécifier une architecture logicielle typée. Ces deux modèles nommés respectivement **modèle de type** et **modèle logique** permettent de séparer la spécification des invariants du système de sa dynamique. Dans SafArchie, les cinq éléments qui composent une architecture logicielle sont le composant primitif, le composite, le port, l'opération et la liaison.

Le *composant* est une entité logicielle définie par l'intermédiaire de son interface avec l'extérieur en vue de sa composition et de sa réutilisation. Il peut posséder des attributs qui caractérisent l'état de ses instances. L'interface du composant met en évidence les opérations que le composant fournit et requiert. Chaque opération est rendue accessible par l'intermédiaire d'un port. Le *port* est l'unité structurelle de composition entre composants. Il est le point d'accès au composant et est potentiellement composé de deux types d'opérations, les opérations fournies et les opérations requises. Un port regroupe une liste d'opérations possédant un lien conceptuel entre elles. Une *opération* définit les structures de données échangées dans une interaction entre deux composants.

Le mode de communication dans SafArchie est le mode synchrone : l'invocation d'une opération requise est bloquante. Ce mode consiste en l'envoi d'une requête suivi de la réception d'une réponse. À partir de ce mode de communication, nous décrivons des modes de communication plus évolués à l'aide de composants dédiés que certains appellent connecteurs.

Au niveau du modèle de type, les *liaisons*, appelées *bindings*, matérialisent les interactions possibles entre composants. Chaque liaison doit relier deux types de port compatibles.

La configuration SafArchie

SafArchie repose sur un modèle de composant hiérarchique. Un composant peut contenir d'autres composants. Il est alors défini comme étant un composite. Un *composite* a une double spécificité : en tant que composant, il peut lui-même avoir un rôle dans un assemblage entre composants. En tant qu'entité contenant d'autres composants, il connaît la configuration des composants dont il a la charge, c'est-à-dire leurs interactions. Les composants contenus au sein d'un composite sont appelés ses fils. Un composant participant à un assemblage appartient nécessairement à un composite qui est appelé le père de ce composant. Au niveau du modèle de type, un type de composite définit à la fois les types de composant qu'il peut contenir, leur cardinalité et leurs liaisons.

Dans le modèle de composant de SafArchie, il existe une différence fondamentale entre les composants primitifs et les composants composites. En effet, outre le fait que les composants primitifs sont des entités opaques (boîtes noires), alors que les composites sont facilement introspectables (boîtes blanches), les composants primitifs sont une unité de traitement fonctionnelle alors que les composites sont une unité de structuration et de

configuration architecturale. C'est pourquoi il existe deux catégories de port, les ports des composants primitifs (*Primitive Port*) et les ports des composants composites (*Delegated Port*). Ces derniers sont des ports délégués. Un composite ne possède pas de ports natifs contenant un traitement associé, il ne fait que déléguer les traitements à un composant par l'intermédiaire des ports délégués. Le port délégué fait donc toujours référence à un port d'un de ses fils à qui il transmet la réalisation de l'interaction. Cette hiérarchie entre les ports peut être étendue à plusieurs niveaux.

Cette distinction entre *port primitif* et *port délégué* se retrouve au niveau du modèle de type d'architecture logicielle. La notion de type de port primitif (*PrimitivePortType*) est quant à elle toujours associée à un type de composant. Un type de port définit la liste des opérations fournies et requises par le type de port.

La notion de composite permet ainsi d'obtenir une représentation abstraite d'une architecture logicielle de granularité plus importante. Contrairement à certains modèles de composant comme Fractal, dans *SafArchie* le composite est une unité d'encapsulation fermée. Un composant ne peut appartenir qu'à un seul composite. Il n'y a pas de partage de composants entre plusieurs composites, ce qui garantit la propriété d'encapsulation.

2.4.5.4 COSA

COSA (*Component-Object based Software Architecture*) [Smeda 2006] [Oussalah *et al.* 2004], développé par l'équipe MODAL[9], est défini comme une approche de description d'architecture basée sur la description architecturale et la modélisation par objets. Dans COSA, les composants, les connecteurs et les configurations sont des classes qui peuvent être instanciés pour établir différentes architectures. Ils peuvent être réutilisés, redéfinis et évolués efficacement par des mécanismes opérationnels bien définis tels que l'instanciation, l'héritage ou la composition. Les mécanismes utilisés dans COSA sont inspirés des mécanismes du paradigme objet. Ces mécanismes sont : l'instanciation, l'héritage et la composition.

Connecteur COSA

COSA confère aux connecteurs un niveau de granularité et de réutilisation semblable à celui des composants. Un connecteur est principalement défini par une interface et une glu. En principe, l'interface décrit les informations nécessaires du connecteur, y compris le nombre de rôles, le type de services fourni par le connecteur et le mode de connexion. Un rôle est soit de type requis ou de type fourni. Le nombre rôles d'un connecteur représente le degré d'un type de connecteur. La glu décrit les fonctionnalités attendues d'un connecteur primitif.

Dans COSA, il y a trois types d'associations qui relient les différents éléments architecturaux : *Attachment*, *Binding* et *Use*. Les *Attachments* permettent de relier les ports d'un composant ou d'une configuration aux rôles d'un connecteur. Un *Binding* fournit une

[9] MODAL : "langages de MODélisation d'Architectures Logicielles" équipe de recherche au laboratoire LINA, Université de Nantes, France

association entre ports (respectivement rôles) internes et les ports (respectivement rôles) externes des composants composites (respectivement des connecteurs composites). L'association *Use* relie des services aux ports (ou aux rôles pour les connecteurs).

Configuration COSA

Les configurations dans COSA sont des entités de première classe. Elles représentent un graphe de composants et de connecteurs. Une configuration peut avoir zéro ou plusieurs interfaces définissant les ports et les services de la configuration. La définition des configurations comme des classes instanciables permet la construction de différentes architectures du même système.

Dans COSA, une architecture peut être déployée de plusieurs manières, sans réécrire le programme de configuration/déplacement. Un autre avantage de COSA est la définition explicite des connecteurs et le support fort de leur réutilisation. Les fonctionnalités des connecteurs et des configurations sont exprimées par les services. Ils sont représentés au niveau interface. Ceci facilite la recherche d'un connecteur ou d'une configuration dans une bibliothèque.

2.4.6 Synthèse sur la modélisation des connecteurs

Le support fourni par les ADLs pour la modélisation des connecteurs est considéré comme moins important que celui des composants. Les ADLs comme Darwin, MetaH, Fractal, Koala, AADL et Rapide ne considèrent pas les connecteurs comme des entités de première classe, mais plutôt les modélisent implicitement. Leurs connecteurs sont toujours définis comme des instances qui ne peuvent pas être manipulées lors de la conception ou réutilisés par la suite. Dans l'ensemble, leur support pour les connecteurs est négligeable, comme nous pouvons l'observer dans le Tableau 2.1. Par contre, tous les ADLs qui modélisent les connecteurs et leurs interfaces explicitement font une séparation claire entre les types de connecteurs et leurs instances. Il est intéressant de noter que le support fourni pour l'évolution et pour les propriétés non-fonctionnelles est rare et que Aesop est le seul ADL qui fournit un support important pour chaque caractéristique de classification. Les Tableaux 2.1 et 2.2 contiennent un sommaire plus complet sur la modélisation des connecteurs par les ADLs.

ADL	Propriétés des connecteurs				
	Interface	Types	Contraintes	Evolution	PNF
Darwin (binding)	implicite ; pas modélisation explicite de l'interaction entre composants	rien ; « composant de connexion »	---	---	---
SafArchie (liaison)	implicite ; pas modélisation explicite de l'interaction entre composants	rien ; « composant de connexion »	---	---	---

Fractal (liaison)	interfaces *cliente*, *serveur*	rien ; « composant de connexion »	contraintes sur les liaisons entre composant	---	---
Koala	interfaces fournies / requises de composants	---	---	---	---
AADL	pas d'interface	aucun ; supporte trois types de port : (*data*, *vent* et *eventData)*	---	---	peuvent être définiées par l'utilisateur mais aucun traitement automatique n'est prévu
UML 2.0	aucune; connexion directe de composants	aucun	contraintes OCL sur les liaisons entre composants	pas de support explicit pour l'évolution	possible, mais uniquement sous forme d'annotations

Tableau 2.1- Modélisation des connecteurs implicites dans les ADLs.

ADL	Propriétés des connecteurs				
	Interface	Types	Contraintes	Evolution	PNF
C2	interface avec chaque composant via des ports séparés ; les éléments d'interfaces sont requis / fourni	système de typage extensible, basé sur les protocoles	chaque port participe à un lien unique	interfaces à contexte réflexif, mécanisme de filtrage évolutif	rien
SOFA	les points d'interfaces sont des rôles	système de typage extensible ;	via les interfaces ;	sous-typage ; raffinement des connecteurs via des patrons génériques	rien
ACCORD	les points d'interfaces sont des prises	système de typage extensible, basé sur les protocoles	Définies par des contrats ;	sous-typages avec préservation du comportement	rien
Wright	les points d'interfaces sont les rôles ; la sémantique d'interaction des rôles est spécifier en CSP	système de typage extensible basé sur des protocoles ; les rôles et la glu sont paramétrables	via les interfaces ; le protocole de chaque rôle est spécifié en CSP	via différents paramètres d'instanciation	rien
COSA	les points d'interfaces des connecteurs sont les « *rôles* »	système de typage extensible basé sur l'héritage	via les interfaces	sous-typage structurel via le mécanisme « *extends* »	possible mais sans aucun traitement
ACME	points d'interfaces « rôles »	typage extensible et paramétrable	via les interfaces	sous-typage structurelle via le mécanisme « *extends* »	possible mais sans aucun traitement
xADL 2.0	modélisée par (schéma, identificateur, description textuelle, et une direction) requise/ fournie	typage explicite (schémas, structures et types)	via les interfaces	sous-typage structurel	possible via des extensions

Tableau 2.2- Modélisation des connecteurs explicites dans les ADLs.

Il est important de noter qu'il y a une faible ressemblance entre les ADLs et les langages d'interconnexion de modules (MILs) [DeRemer et Kron 1975]. Certains ADLs, comme Wright et Rapide, modélisent les composants et les connecteurs à un niveau élevé d'abstraction et n'assurent aucune relation entre une description architecturale et ses implémentations. Ces langages sont désignés comme « *langages à implémentation indépendante* ». Par contre il y a d'autres ADLs (Weaves, UniCon, et MetaH, AADL, l'ADL Fractal) qui exigent un degré beaucoup plus élevé de conformité entre une architecture et ses implémentations. Les composants modélisés dans ces langages sont directement liés à leurs implémentations. Cependant, la spécification des interconnexions des modules ne peut être distinguée de la description architecturale. Cette catégorie de langage est désignée par « *langages à implémentation contraignante* » [Binns *et al.* 1996] [Medvidovic et Taylor 2000].

Les sous-architectures dans xADL 2.0, les conteneurs dans l'ADL Fractal, les composants composites dans C2, les configurations dans Wright et Koala, et les systèmes dans ACME sont des appellations différentes pour le même concept de configuration dans chaque ADL.

2.4.7 Synthèse sur la modélisation des configurations

C'est au niveau des configurations que la valeur ajoutée de certains ADLs peut être facilement notifiée. A titre d'exemple, la contribution particulière de SADL est dans le raffinement architectural, alors que Darwin est principalement axé sur la composition de systèmes. Aucun ADL n'arrive à satisfaire tous les critères qui nous semblent important dans la modélisation de configuration d'architectures logicielles, bien que Rapide et Weaves se rapprochent de cet objectif. La réalisation de ces critères est un objectif toujours recherché dans les ADLs. Les Tableaux 2.3 et 2.4 présentent un sommaire plus complet sur la modélisation des configurations par les ADLs.

ADL	Propriétés des configurations				
	Composition	Raffinement et Traçabilité	Passage à l'échelle	Contraintes	Evolution
Darwin	fournie via les composants composites du langage	supporte la génération de systèmes avec implémentation contraignante	entravé par les configurations en-lignes	services fournis doivent être reliés uniquement aux services requis	pas de support à cause des configurations en-lignes
SafArchie	fournie via les ports composants composites du langage	via les liens entre les interfaces de même type (*binding*)	entravé par des connexions implicites	Ports fournis doivent être reliés uniquement aux ports requis	pas de support à cause des configurations en-lignes
Fractal	Fournie via les conteneurs et contenus (composite)	Via les liaisons entre les interfaces internes du composant et les interfaces externes des sous-composants	possible vie la récursion	les interfaces fournies doivent être reliés uniquement aux interfaces requises	possible via les template , *geniricfactory* et la reconfiguration dynamique
Koala	composition hiérarchique	via les liens entre les interfaces de même type (*binding*)	via les paramètres de configuration	impossible de changer les interfaces existantes	modélisation des lignes de produit avec des éléments optionnels et variables

AADL	fournis via les concepts de systèmes et composants abstraits	---	entravé par des connexions implicites	---	pas de relation « *extends* » entre les systèmes et composants abstraits
UML 2.0	fournis via relations de composition d'association et délégation	diagramme de structures composites	extensible via les profils	interface fournie doit être attachée à une interface requise en plus des contraintes OCL	pas de support pour modéliser l'aspect évolution

Tableau 2.3 : Modélisation en-lignes des configurations dans les ADLs

ADL	Propriétés des configurations				
	Composition	Raffinement et Traçabilité	Passage à l'échelle	Contraintes	Evolution
C2	supporté via l'architecture des composants internes	pas de raffinement	aidé par les configurations explicites et le nombre variable de ports de connecteurs	des invariantes stylistiques fixes	aidé par les configurations explicites ; peu d'interdépendance entre composant ; connecteur hétérogènes
SOFA	supporté via l'architecture des composants internes	---	aidé par les configurations explicites ;	Les ports doivent êtres attachés uniquement aux rôles	seulement les composants peuvent être mis à jours dynami-quement (sans les connecteurs) ;
ACCORD	---	---	---	----	---
Wright	les fonctions et la glu de composites sont représentées en architectures	pas de raffinement	soutenu par les configurations et les sockets explicites et le nombre variable de rôles ; utilisé dans des projets à grande échelle	les ports doivent être attachés uniquement aux rôles	adaptée pour la spécification partielle; configurations explicites
COSA	fournie via les interfaces de configurations	---	soutenu par les configurations explicites ;	les ports doivent être attachés uniquement aux rôles et inversement	pas de support pour les architectures partielles ; soutenu par les configurations explicites
ACME	fournis via templates, representation et rep-maps	rep-maps	assisté par des configurations explicites; entravé par nombre fixe de rôles	Les ports doivent êtres attachés uniquement aux rôles	possible via des configurations explicites ; familles de première classe
xADL 2.0	via un mécanisme de composition	---	les options, les versions et les variantes d'éléments architecturaux	contraintes via les expressions « Gards » et « Boolean Gards »	modélisation de l'évolution

Tableau 2.4 : Modélisation explicite des configurations dans les ADLs

2.5 Conclusion

La principale contribution de ce chapitre est la définition d'un cadre de comparaison des langages de description d'architectures. Ce dernier est défini par deux principaux axes, représentant les concepts clés de notre travail, à savoir les connecteurs et les configurations. Notons que les architectures logicielles se distinguent des autres notations par l'intérêt explicite qu'ils accordent aux connecteurs et aux configurations architecturales. Sur chaque axe nous avons fixé un ensemble de caractéristiques que nous jugeons déterminant pour chaque concept.

La remarque la plus surprenante que nous pouvons relever dans cette étude comparative est l'inconsistance avec laquelle les ADLs supportent les connecteurs après avoir déterminé leur rôle dans les descriptions architecturales. Plusieurs ADLs fournissent un minimum de supports pour la modélisation des connecteurs. D'autres permettent uniquement la modélisation de connecteurs complexes (exemple de Wright) ou l'implémentation des connecteurs simples. Nous avons effectué cette analyse dans le but de mettre en évidence les insuffisances de chaque ADL suivant les deux axes fixés.

D'après les résultats présentés dans les Tableaux 2.1, 2.2, 2.3 et 2.4 nous avons établi le constat suivant :

1- une faible prise en charge des connecteurs où très peu d'ADLs les considèrent comme entités de première classe au même niveau conceptuel que les composants,

2- la plupart des langages de description ne traitent pas le concept de configuration explicitement,

3- la description hiérarchique avec plusieurs niveaux de descriptions imbriquées est mal supportée,

4- peu d'importance est accordée à la relation de conformité entre une architecture et ses implémentations.

Dans le chapitre 3 nous allons proposer notre méta-modèle pour la description des architectures logicielles. Les insuffisances que nous avons pu relever sur les langages de description architecturale dans ce chapitre, feront l'objet des motivations de notre méta-modèle.

CHAPITRE 3

C3 : un méta-modèle pour la description des architectures logicielles

3.1 Introduction

L'objectif de l'architecture logicielle est d'offrir une vue d'ensemble et un fort niveau d'abstraction afin d'être en mesure d'appréhender les systèmes logiciels qui sont devenus de plus en plus complexes [Garlan et Shaw 1993]. Dans le cadre de cette thèse, nous nous intéressons spécialement aux architectures logicielles à base de composants [Bass *et al.* 1998]. Ces dernières reposent sur une décomposition du système en un ensemble de composants (*unités de calcul ou de stockage*) qui communiquent entre eux par l'intermédiaire de connecteurs (*unités d'interaction*).

Les concepteurs de logiciels ont besoin de langages de description d'architectures extensibles et de mécanismes flexibles permettant de manipuler les éléments et les concepts de base utilisés dans la description de l'architecture. Une architecture logicielle est construite à partir de composants, de connecteurs et de configurations avec des contraintes sur leur assemblage ainsi que d'autres contraintes sur le comportement des composants et des connecteurs.

Les ADLs de première génération ont tous modélisé les caractéristiques structurelles et fonctionnelles des systèmes logiciels, à l'exception d'ACME [Garlan *et al.* 2000] qui s'est essentiellement intéressé à la vue structurelle. Ces ADLs traitent principalement chaque vue de l'architecture logicielle indépendamment des autres. Dans ce chapitre, nous introduisons d'autres points de vue portant sur de différentes préoccupations architecturales qui sont complémentaires à la vue structurelle.

Pour manipuler les éléments de base d'une telle architecture, nous avons besoin de plusieurs mécanismes d'interactions, alors que la majorité des ADLs ne proposent, comme mécanique opératoire, que le sous-typage entre éléments, cas d'ACME [Garlan *et al.* 2000] et de C2 [Taylor *et al.* 1996].

Sur la base d'une large étude portant sur les approches et les langages de notation ou de description d'architectures (cf. Chapitre 2) nous avons identifié que les aspects fondamentaux utilisés pour décrire les architectures logicielles d'un système sont centrés sur les concepts de composant, de connecteur et de configuration.

Dans ce chapitre nous introduisons notre modèle proposé baptisée C3 (*Component, Connecteur et Configuration*) comme un nouveau modèle de description d'architectures logicielles à base de composants, de connecteurs et de configurations. Ce modèle représente une approche descriptive de type *Top-Down*. La contribution principale de cette approche est, d'une part d'emprunter le formalisme des ADLs et de les étendre grâce aux concepts et mécanismes objets, d'autres part expliciter les connecteurs et les configurations en tant qu'entités de première classe au même niveau conceptuel que les composants. Nous présentons également le modèle MY comme une démarche pour guider le concepteur dans son processus de modélisation d'architecture logicielle. MY décrit les concepts d'architecture logicielle selon trois branches : composant, connecteur et configuration.

Le reste de ce chapitre est organisé comme suit : La section 3.2 décrit le méta-modèle C3. Dans la section 3.3, nous allons introduire le concept d'architecture physique. La section 3.4 introduit un bilan sur le méta-modèle C3. Un formalisme pour la représentation des concepts de C3 sera présenté dans la section 3.5. La dernière section conclut ce chapitre.

3.2 Le méta-modèle C3

Dans la conception de C3, nous avons cherché à réaliser une séparation stricte entre les concepts de composant, de connecteur et de configuration. Les constructeurs de composants ne font pas d'hypothèses sur les configurations dans lesquelles leurs composants doivent être utilisés. De même, les concepteurs de configurations ne sont pas autorisés à changer le fonctionnement interne d'un des composants pour l'adapter à leur configuration. Aussi, nous avons cherché à encapsuler uniquement la fonction d'interaction dans les connecteurs, la fonction de traitement et de calcul dans les composants et la description de la topologie dans les configurations. Nous décrivons en détail ces concepts dans la section suivante.

Définir un modèle de description est la partie importante de notre travail. Cependant, il est important d'assurer l'efficacité d'un tel modèle. Pour cela, nous y avons associé deux propriétés importantes :

- *la minimalité :* l'idée basique derrière cette propriété est de fournir un ensemble minimal et suffisant de concepts dans le modèle de description, ainsi appelé « noyau ». En règle générale, un nombre réduit de concepts raccourcit la courbe d'apprentissage.
- *la complétude :* étroitement liée à la minimalité, la complétude permet d'étendre le noyau sans perturber la façon dont il fonctionne.

En plus de ces deux propriétés, nous visons à définir un modèle générique. Cela passe par la méta-modélisation. En effet, la définition d'un méta-modèle permet de ne pas limiter notre approche à un simple langage ou une notation de modélisation. L'approche se place à un niveau d'abstraction supérieur aux langages et notations de modélisation. Nous nous

intéressons principalement aux concepts utilisés pour décrire les architectures logicielles. Il s'agit de réifier ces concepts et de représenter explicitement leurs caractéristiques et les relations qui existent entre eux. Ainsi, l'approche s'abstrait de toute syntaxe et manipule les éléments de modélisation comme des entités sémantiques et non des entités syntaxiques.

Afin de mieux construire notre méta-modèle C3, nous avons défini principalement deux modèles complémentaires (cf. Figure 3.1). Le premier est considéré comme le noyau de C3 et défini la *représentation* des concepts architecturaux. Le deuxième est un modèle de *raisonnement* sur l'architecture logicielle [Amirat et Oussalah 2008]. Nous utilisons le modèle de représentation pour décrire les architectures en se basant sur les éléments de C3 et nous utilisons le modèle de raisonnement pour comprendre et analyser les architectures logicielles.

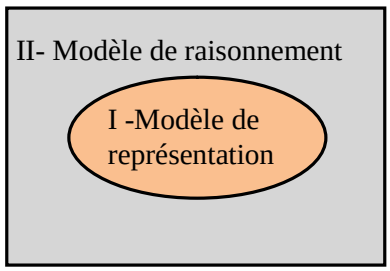


Figure 3.1- Les modèles de définition de C3.

3.2.1 Modèle de représentation

Les éléments de base du modèle de représentation de C3 sont les composants, les connecteurs, et les configurations. Chacun de ces éléments possède une interface d'interaction avec son environnement. Les concepts de base de notre méta-modèle C3 sont décrits à travers la Figure 3.2. Nous nous appuyons sur une modélisation objet pour la description du méta-modèle. Nous utiliserons précisément le formalisme du diagramme de classes de la notation UML 2.0. Ainsi, les concepts clés de classe, d'héritage, de composition, d'association et multiplicités sur les associations sont exploités. Le diagramme de classe met en évidence les concepts proposés par C3, ainsi que les associations entre eux. Dans les sections suivantes, nous apportons une définition pour chaque élément architectural de C3 ainsi qu'un exemple illustratif quand cela est nécessaire.

3.2.1.1 Les composants dans C3

Un composant est une unité de calcul ou de stockage de données. Il peut être aussi petit qu'une simple procédure ou aussi grand qu'une application entière. Un composant peut exiger ses propres données de traitement ainsi que son propre espace d'exécution, comme il peut les partager avec d'autres composants [Medvidovic *et al.* 1999].

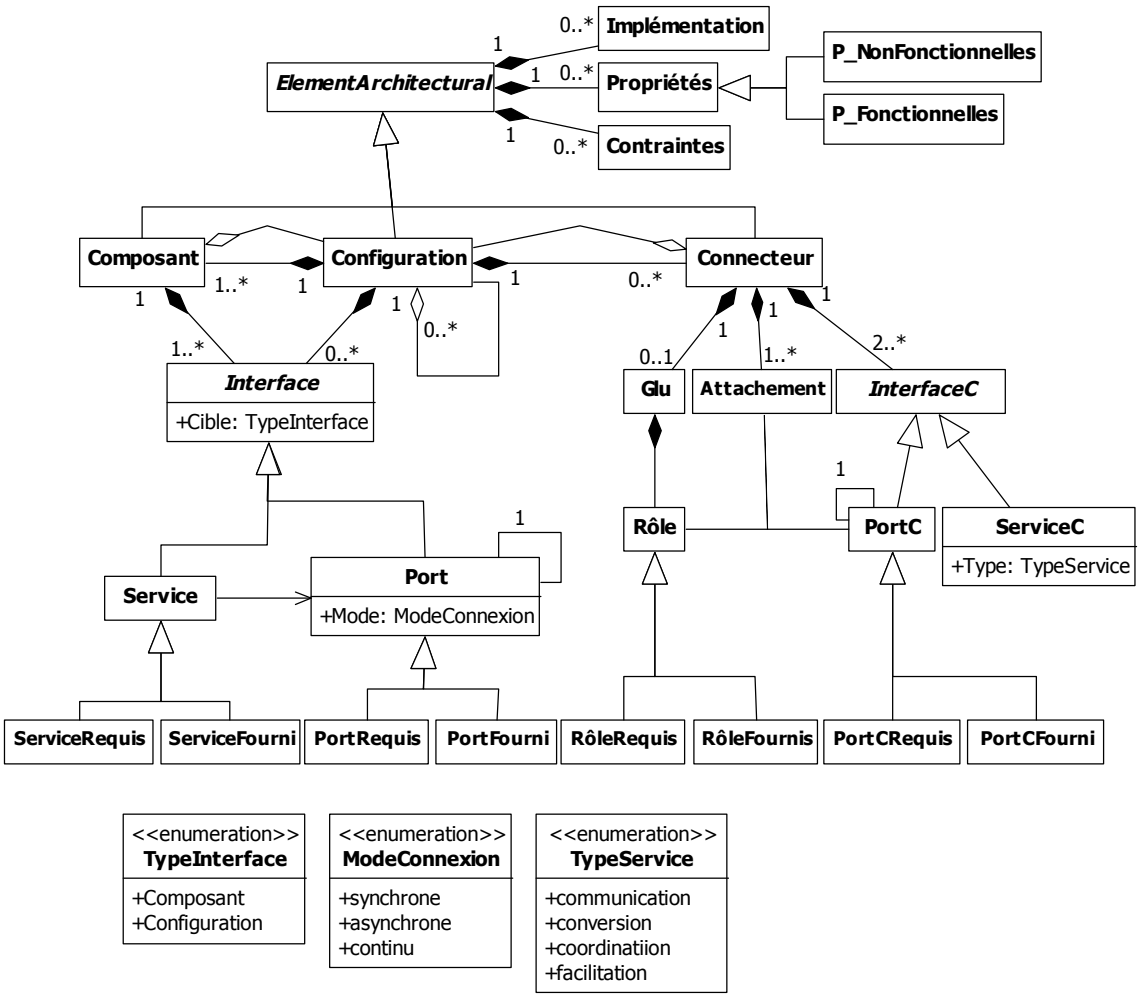


Figure 3.2- Éléments de base du méta-modèle C3.

Afin d'être en mesure de raisonner sur un composant et sur l'architecture qui l'englobe, chaque composant doit fournir la spécification de ses besoins en entrée ainsi que les services qu'il doit rendre à son environnement. Dans C3 chaque composant possède sa propre interface qui spécifie ses services requis et fournis ainsi que les contraintes d'utilisation de ces services. Les entrées et les sorties de services d'un composant passent par des points d'interactions. Chaque point d'interaction d'un composant est appelé *Port*. Les ports sont nommés et typés. Nous faisons une distinction claire entre un port requis et un port fourni. Chaque port peut être utilisé par un ou plusieurs services suivant le protocole associé.

La sémantique d'un composant est modélisée afin de rendre possible son évolution, son analyse, l'application de contraintes sur son utilisation et la consistance de la correspondance des architectures entre les différents niveaux d'abstraction. La spécification de la structure d'un composant primitif est substituée par la spécification de ses ports requis et fournis. La fonction d'un composant est spécifiée par les services échangés, en entrée ou en sortie, avec son environnement.

Dans cette thèse, nous considérons un composant comme étant une unité de conception et de composition réutilisable qui spécifie de manière explicite ses services fournis ou requis à travers ses interfaces. Nous nous limitons à cette simple définition qui est le résultat de la fusion des deux définitions ci-dessus. Nous ne nous intéressons pas dans ce travail à l'aspect de déploiement des composants. La Figure 3.3 décrit un type composant d'une architecture C3.

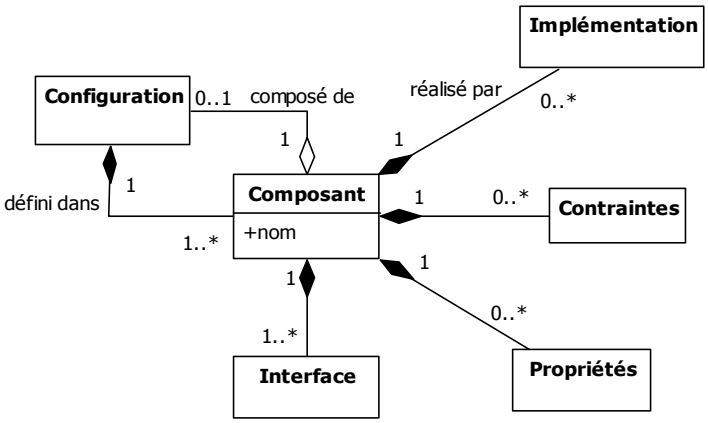


Figure 3.3- Structure d'un composant C3.

Selon cette description la spécification de la signature d'un composant type C3 est définie comme suit :

```
Composant composantType (interfaceRequise, interfaceFournie) // signature d'un composant
{
   Propréités = { Liste des propriétés d'un composant }
   Constraintes = { liste des contraintes à respecter }
   Computation = { partie calcul d'un composant }
}
interfaceRequise = {ListePortsRequis} U {ListeServicesRequis}
interfaceFournie = {ListePortsFournis} U {ListeServicesFournis}
```

3.2.1.2 Les connecteurs dans C3

Un connecteur est essentiellement représenté par une *interface* et une spécification de la *glu* [Oussalah *et al.* 2005]. Principalement, l'interface affiche les informations nécessaires du connecteur, notamment le nombre de points d'interactions, le type des rôles, le type des services fournis par un connecteur (communication, la conversion, la coordination, la facilitation), le mode de connexion (synchrone, asynchrone), le mode de transfert (parallèle, série), etc. La glu d'un connecteur représente le protocole utilisé pour gérer les liaisons entre les rôles d'entrée et les rôles de sortie d'un connecteur.

3.2.1.2.1 Définition

Dans le méta-modèle C3, les points d'interaction d'une interface sont appelés *ports*. Un port est l'interface d'un connecteur destiné à être liée à une interface de composant (un port de composant). Dans notre contexte, les ports sont typés. Ils peuvent être de type requis ou fournis. Un port fourni de connecteur est destiné à être connecté à un port requis de composant, et un port requis de connecteur est destiné à être connecté à un port fourni de composant. Le nombre de ports dans un connecteur dénote le degré de son type. Par exemple, dans une architecture client-serveur un connecteur de type RPC[1] représentant l'interaction entre trois clients et un serveur est un connecteur de degré quatre.

Des interactions plus complexes entre plusieurs composants sont habituellement implémentées par des types de connecteurs de degrés plus élevés. Par conséquent, l'interface est la partie visible du connecteur, elle devra contenir suffisamment d'informations concernant les types des ports et des services que le connecteur peut supporter. Ainsi, le concepteur peut décider, à partir d'une simple consultation de l'interface, si un tel connecteur peut répondre à ses besoins ou non.

La *glu* décrit la fonctionnalité que le connecteur devra fournir à son environnement. Elle représente la partie cachée d'un connecteur. La glu peut être juste un simple protocole de liaison entre ports d'entrées et de sorties, comme il pourrait s'agir d'un protocole complexe réalisant diverses opérations, notamment la liaison, la conversion de format de données, le transfert, l'adaptation, etc. D'une manière plus globale, la glu d'un connecteur représente le type de connexion que celui-ci offre dans une interaction entre composants.

Les connecteurs peuvent également avoir d'autres fonctionnalités, comme le calcul et le stockage de l'information. Par exemple, un connecteur peut être doté d'un algorithme de conversion des données d'un format « *a* » vers un format « *b* » ou d'un algorithme chargé de compresser les données avant de les transmettre à nouveau, etc. . Par conséquent, le service fourni par les connecteurs est défini par sa glu.

Les connecteurs dans C3 sont des entités architecturales *primitives* ou *composites*. Un connecteur primitif est un connecteur qui n'a pas de structure interne explicite ceci dit qu'il ne peut pas être décomposé. Généralement, il est conçu pour modéliser une interaction simple. A l'opposé, un connecteur composite est un connecteur qui possède une structure interne (*configuration*) composée à partir d'un ensemble de composants et de connecteurs. Cette catégorie de connecteurs est conçue pour modéliser les interactions complexes dans une architecture. Un connecteur composite possède également une interface représentant ses ports et ses services mais il ne possède pas de glu ; sa structure interne est décrite par une architecture.

3.2.1.2.2 Structure d'un connecteur

Notre contribution à ce niveau consiste dans l'amélioration de la structure des connecteurs en encapsulant les liens d'attachements à l'intérieur du connecteur lui-même (cf.

[1] *RPC : Remote Procedure Call*

Figure 3.4). Ainsi, le concepteur d'architectures n'aura aucun effort à dépenser dans la description des liens d'attachements entre son connecteur et les composants compatibles, parce que les liens sont déjà décrits à l'intérieur du connecteur. Par conséquent, la tâche du concepteur consiste seulement à choisir, parmi les éléments de la bibliothèque, le type du connecteur approprié où son interface est compatible avec les interfaces des composants qui sont censés être assemblés avec ce dernier [Amirat *et al.* 2007].

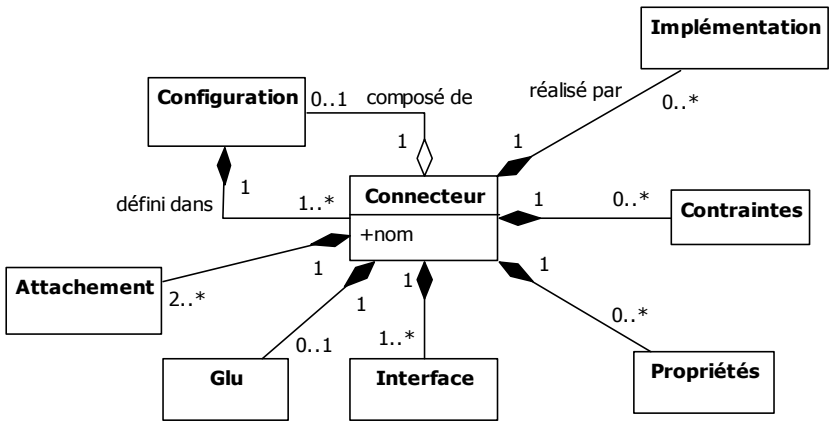


Figure 3.4- Structure d'un connecteur C3.

Dans C3, la signature générique d'un connecteur type dans le méta-modèle C3 est définie comme suit :

```
Connecteur ConnecteurType (interfaceRequise, interfaceFournie) // signature d'un connecteur
{
   Propréités = { Liste des propriétés d'un composant }
   Constraintes = { liste des contraintes à respecter }
   Services = { Liste des services }
   Glu = { protocole d'interaction entre rôles }
   Attachements = { Liste des attachements entre ce connecteur et son envoronnement }
}
interfaceRequise = {ListePortsRequis} U {ListeServicesRequis}
interfaceFournie = {ListePortsFournis} U {ListeServicesFournis}
```

Afin d'illustrer les propriétés du connecteur C3, nous utilisons une version simplifiée de l'exemple client-serveur pour illustrer nos propos. L'architecture de cet exemple est décrite à partir d'une configuration client-serveur « *CS-config* » (cf. Figure 3.5). A l'intérieur de cette configuration nous avons deux composants ; le composant *client* (composant primitif) et le composant *serveur*. Ces deux composants sont reliés entre eux par un connecteur de type RPC, qui spécifie les interactions qui peuvent avoir lieu entre le *client* et le *serveur*. Dans la Figure 3.5, ce type de connecteurs est représenté par une ligne continue.

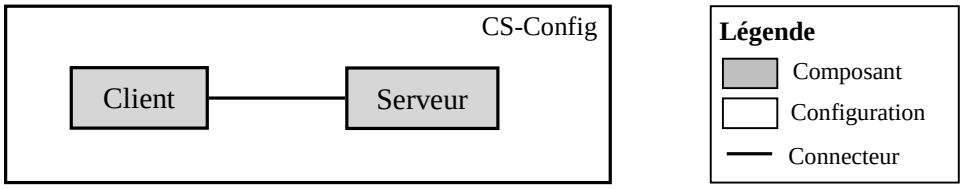


Figure 3.5 - L'architecture Client-Serveur.

La Figure 3.6 illustre la structure du connecteur « RPC » reliant le composant client « C » avec le composant serveur « S ». Suivant la nouvelle structure de connecteur que nous proposons, le connecteur « RPC » encapsule les attachements qui représentent les liens d'attachements entre le client et le serveur. Ces liens d'attachements portent la même sémantique que celle évoquer par ACME.

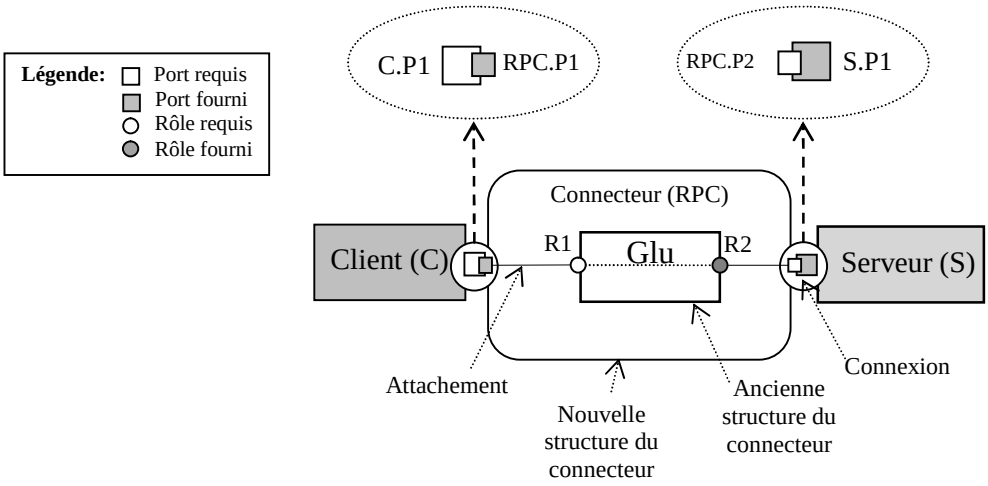


Figure 3.6- La nouvelle structure d'un connecteur dans le méta-modèle C3.

Dans la Figure 3.7 nous utilisons une syntaxe proche d'ACME pour décrire la spécification du connecteur « RPC » déployé dans l'architecture client-serveur. A l'intérieur de ce type de connecteur nous avons le code de la glu décrivant comment les activités du client et du serveur sont coordonnées. Ce code indique que les activités doivent être séquencées dans un ordre bien défini: le client demande un service, le serveur traite la requête et fournit le résultat que reçoit le client.

```
Connecteur  RPC ( C.P1, S.P1  )    // signature du connecteur RPC
{
   Propréités  = { Liste des propriétés }
   Constraintes = {  Niveau Hiérachique : (niveau(C) = niveau(S) }  // niveau de décomposition
   Services = { Liste des services }
   Glu = {Rôles =( (R1 , R2) ; R1 = R2 ) }        // simple cas d'une glu
   Attachements = { R1  à  C.P1, R2  à  S.P1 }    // attachements
}
```

Figure 3.7- Spécification du connecteur RPC en C3.

Par l'encapsulation des attachements à l'intérieur du connecteur et avec une spécification exacte de la signature, l'interface identifie les différents types de composants qui peuvent être reliés par un type donné de connecteurs. De cette manière, les composants et les configurations sont assemblés de manière simple, uniforme et cohérente, sous la forme d'un puzzle architectural ou un « *Lego Blocks* » sans effort à fournir pour décrire les liens entre les composants et connecteurs. L'architecture résultat est supposée être correcte du moment que les attachements sont déjà décrits à l'intérieur des connecteurs. Cette approche permet de réduire le temps de développement de systèmes à base de composants, de faciliter leurs évolutions, de maintenir leur cohérence et de promouvoir le marché des composants (COTS[2]).

3.2.1.3 Les configurations dans C3

Les configurations de C3 sont des entités de première classe. Elles représentent un graphe de composants et de connecteurs. Dans C3, une propriété principale de description de système est que la topologie globale d'un système est définie indépendamment des composants et des connecteurs formant le système. Ainsi, nous considérons une configuration comme une classe qui peut être instanciée plusieurs fois pour donner plusieurs architectures d'un système.

Dans C3 une configuration est aussi réutilisable comme tous ses constituants internes. Bien que les instances de composants et de connecteurs soient encapsulées dans des configurations, leurs types correspondant ne le sont pas. Par conséquent, il est possible de construire d'autres configurations avec les mêmes éléments suivant des assemblages différents. Une configuration peut avoir zéro ou plusieurs interfaces définissant les ports et les services de la configuration. Les ports sont les points de connexions qui seront reliés aux ports des composants internes ou aux ports des éléments externes. Les configurations sont structurées de manière hiérarchique : les composants et les connecteurs peuvent représenter des sous-configurations qui disposent de leurs propres architectures. La Figure 3.8 définit les principales parties d'une configuration dans C3.

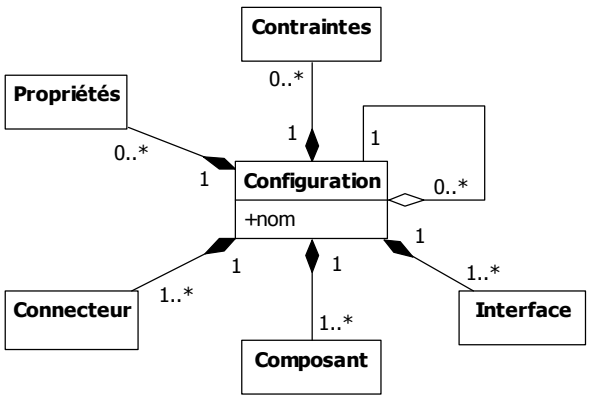


Figure 3.8- Structure d'une configuration C3.

[2] *COTS*: *Components off The Shelf*

Selon cette description la spécification de la signature d'une configuration type dans C3 est définie comme suit :

Configuration configurationType (*interfaceRequise, interfaceFournie*)
{
Propréités = { Liste des propriétés d'une configuration }
Constraintes = { liste des contraintes à respecter }
ElémentsInternes = { liste des composants et des connecteurs formant la configuration }
Topologie = {Liste des interconnections entre les composants et connecteurs internes}
}
interfaceRequise = {ListePortRequis} U {ListeServiceRequis}
interfaceFournie = {ListePortFourni} U {ListeServiceFourni}

3.2.1.4 Interface dans C3

Chaque élément architectural dans C3 possède une interface associée à un type d'éléments qui correspond à l'ensemble des opérations qu'il définit. Via cette interface, l'élément publie ses services pour l'environnement extérieur. Les services peuvent être requis et/ou fournis. Toutefois, les éléments sont sélectionnés et connectés à partir de leurs points d'interaction publiés par l'interface.

Pour établir des connexions entre éléments, nous utilisons les ports requis/fournis des composants et les rôles requis/fournis des connecteurs. Une fois ces connexions établies, l'assignation de services aux ports et aux rôles sera possible afin de les exploiter en respectant les contraintes définies au niveau de chaque élément. D'un point de vue conceptuel les ports et les services sont des classes concrètes dérivées de la classe abstraite interface comme le montre la Figure 3.2.

Par ailleurs, la cardinalité est utilisée pour décrire la multiplicité de chaque liaison (*connexion*) entre les éléments architecturaux. Pour une liaison donnée la cardinalité exprime le nombre de ports ainsi que le nombre de rôles qui participent à cette liaison [Oussalah *et al.* 2007].

Les concepts architecturaux que nous avons présentés dans les sections précédentes sont manipulés et exploités via des mécanismes prédéfinis dans le modèle de raisonnement. Plus précisément, dans ce modèle, nous étudierons les différents mécanismes de connexions. Dans la section suivante, nous définirons le cadre et la sémantique d'utilisation de chaque mécanisme.

3.2.2 Modèle de raisonnement

Dans notre approche, nous avons planifié d'analyser l'architecture logicielle à partir de plusieurs points de vue liés à des hiérarchies de dépendance distinctes. Chaque hiérarchie possède différents niveaux de représentation. La Figure 3.9 illustre les points de vue du

modèle de raisonnement de C3. Ce modèle est défini par quatre types de hiérarchies de dépendance, chaque type représente une vue particulière du modèle de représentation.

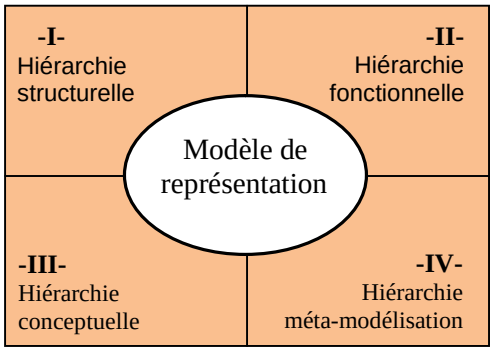


Figure 3.9- Modèles de représentation et de raisonnement pour C3.

Les quatre hiérarchies de dépendance sont:

1. La hiérarchie *structurelle*, pour expliciter les différents niveaux de structures complexes et imbriquées qu'on peut avoir dans un système.
2. La hiérarchie *fonctionnelle*, pour établir les différents niveaux hiérarchiques des fonctions d'un système.
3. La hiérarchie *conceptuelle*, pour décrire les bibliothèques de types d'éléments correspondants aux éléments structurels ou fonctionnels utilisés pour décrire l'architecture logicielle.
4. La hiérarchie de *méta-modélisation*, pour établir les liens de correspondance entre les différents niveaux de modélisation (instance, modèle, méta-modèle et méta-méta-modèle) à l'instar des différents niveaux de modélisation de la pyramide des modèles définis par l'OMG (*Object Management Group*) [OMG 2005] [OMG 2007].

En ce qui concerne notre modèle C3, nous distinguons deux types de représentation pour une architecture donnée. La première concerne la représentation externe de l'architecture type (*Architecture Logique*) telle qu'elle est perçue par le concepteur. Tandis que la deuxième représente la forme interne (*Architecture Physique*) de l'architecture instanciée de l'architecture logique. Dans les sections suivantes, nous présenterons les différentes hiérarchies relatives à l'architecture logique et nous étudierons la représentation des niveaux d'abstraction dans chaque type de hiérarchie, avec les mécanismes de connexion associés. Par contre, l'architecture physique sera présentée uniquement dans la section 3.3.

Pour d'illustration, nous allons étudier l'architecture d'un système client-serveur définie précédemment mais avec plus de détails (cf. Figure 3.10). Les sections suivantes illustrent des vues fournies par ces quatre types de hiérarchies.

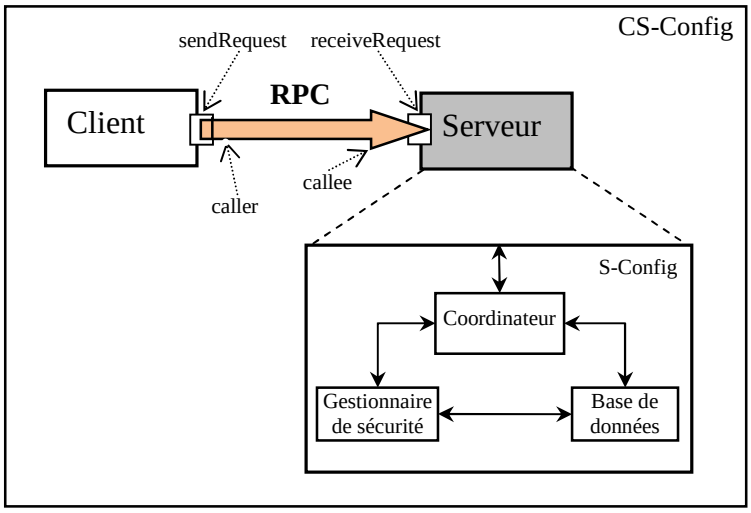


Figure 3.10- Architecture détaillée de l'exemple Client-Serveur.

Dans les figures suivantes nous utilisons des numéros pour représenter les éléments de l'architecture client-serveur. Ces numéros seront comme suit :

(1) : configuration client-serveur,
(2) : composant client,
(3) : configuration serveur,
(4) : composant coordinateur,
(5) : composant gestionnaire de sécurité,
(6) : composant bases de données.

3.2.2.1 Hiérarchie structurelle (HS)

La hiérarchie structurelle représente la structure globale (*topologie*) d'un système en termes d'éléments architecturaux, définis dans le langage de description d'architectures. Plusieurs ADLs académiques comme Aesop, MetaH, Rapide, SADL [Matevska-Mayer *et al.* 2004] ou des ADLs industriels tels que EJB/J2EE [EJB 2003] [Eclipse 2007], CCM/CORBA [CCM 2002] [Pinto *et al.* 2005] ne permettent que la description plate d'architectures. En utilisant ces ADLs l'architecture est décrite uniquement en termes de composants reliés par des connecteurs sans aucune structure imbriquée. Par conséquent, ces architectures n'ont aucune structure hiérarchique. Ce choix de conception de langage a été fait dans le but de simplifier la description de la structure et aussi par absence de concepts et de mécanismes qui doivent respectivement définir et manipuler des structures imbriquées (*composites*).

Dans notre méta-modèle, la structure des architectures est décrite en utilisant des composants, des connecteurs et des configurations, où les configurations sont des éléments composites. Chaque élément de la configuration (composant ou connecteur) peut être primitif, avec un scénario basique de fonctionnement, ou composite à l'aide d'une nouvelle configuration qui contient un autre ensemble de composants et de connecteurs, et ainsi de suite. On peut avoir aussi des structures hiérarchiques à plusieurs niveaux de compositions

(L_0, L_1,..., L_{n-1}, L_n). Où *n* est déterminé en fonction de la complexité du problème, des outils de support et des choix de conception. Il est important de noter que pratiquement toutes les solutions architecturales des systèmes réels sont de natures hiérarchiques.

L'arbre illustré par la Figure 3.11 représente la hiérarchie structurelle relative à l'architecture client-serveur. Le nœud racine représente le niveau d'abstraction le plus élevé (L_0). Il représente également la configuration globale (*CS-Config*), qui contient tous les éléments de l'architecture du système client-serveur. Les nœuds du niveau (L_1) représentent les composants client et serveur. Les feuilles de l'arbre représentent les composants primitifs. Ainsi, une configuration ne sera jamais représentée par une feuille dans l'arbre d'abstraction. Les arcs représentent les liens entre les éléments père et fils. Dans cet arbre un lien ne représente nécessairement pas un lien de service entre le nœud père et les nœuds fils. Pour passer d'un niveau hiérarchique structurel à un autre, nous utilisons un connecteur de décomposition et de composition structurelle (CDCs). Pour relier les éléments appartenant au même niveau hiérarchique, nous utilisons un connecteur de connexion structurelle (CCs). Pour établir un lien de service entre une configuration et ses éléments internes, nous utilisons un connecteur d'expansion et compression (ECC). Ces trois types de connecteurs seront détaillés dans les sous-sections qui vont suivre [Amirat et Oussalah 2009].

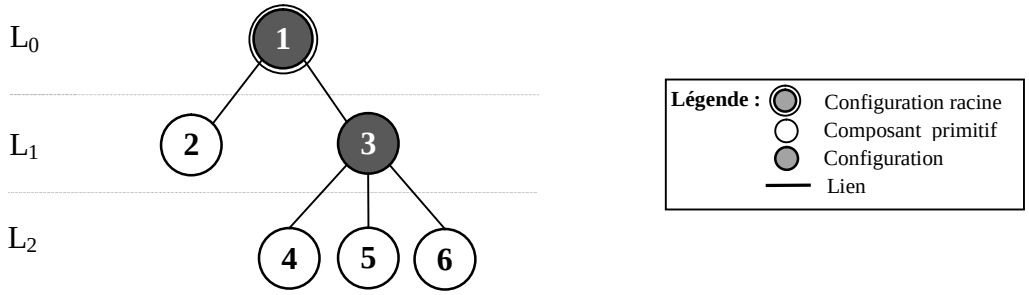


Figure 3.11- Vue externe de la hiérarchie structurelle.

3.2.2.1.1 Connecteur de composition/décomposition structurelle (CDCs)

La composition est un mécanisme qui exige que les éléments aient des interfaces bien définies puisque leurs implémentations sont cachées. Plusieurs ADLs utilisent ce mécanisme pour définir des configurations, où les systèmes sont définis comme des configurations qui sont construites à partir de composants et de connecteurs. Le méta-modèle C3 utilise le mécanisme de composition pour définir des composants composites « *configurations* » de plus en plus complexes, déléguant leurs fonctionnalités aux composants internes. Aussi, ce mécanisme peut être vu comme une opération de décomposition si on opte pour une approche descendante dans la description d'une architecture.

Dans notre méta-modèle, nous proposons un connecteur appelé CDCs (*Connecteur de décomposition composition structurelle*) pour réaliser la décomposition d'une configuration en un ensemble de sous-éléments (opération de raffinement d'une configuration). Aussi, le CDCs peut être utilisé pour composer un ensemble d'éléments pour former un élément composite (opération d'abstraction d'éléments). Un connecteur CDCs modélise donc les relations de décomposition et de composition d'une configuration dans le méta-modèle C3.

// decomposition d'une configuration X en un ensemble de sous-éléments internes Y_i
CDCs decompConnector (X.decomPort , { Y_i .compPort});

// composition de Y_i éléments pour construire une configuration X
CDCs compConnector ({Y_i.compPort} , X.decompPort);

Où X est une configuration, $Y_i \in$ {composant, configuration}, i=1..*n*
$X \in L_k$ et $Y_i \in L_{k+1}$ // niveau(Y_i) = niveau(X) +1
L_k est le niveau hiérarchique numéro k, où k est un entier positif,
Le nombre total de ports de ce connecteur est (*n+1*).

Suivant cette description, un connecteur CDCs peut avoir (*n+1*) ports, où *n* est le nombre d'éléments internes à la configuration à décomposer. Ce type de connecteur permet d'une part, de construire l'arbre hiérarchique des éléments déployés dans une architecture et d'autre part, il permet à une configuration de propager de l'information à tous ses éléments internes sans exception (propagation du haut vers le bas). En revanche, lorsque le CDCs est utilisé pour faire la composition d'un ensemble d'éléments, alors les éléments internes à la configuration peuvent envoyer des informations à leur configuration (propagation du bas vers le haut).

De ce fait, lors de la conception de ce type de connecteur, le concepteur peut choisir d'associer une glu qui correspond à la fonction de décomposition ou à la fonction de composition. De même, le concepteur peut envisager une glu qui gère la propagation dans les deux sens au sein du même connecteur. La Figure 3.12 représente les liens possibles qu'un connecteur de type CDCs peut avoir dans une architecture donnée.

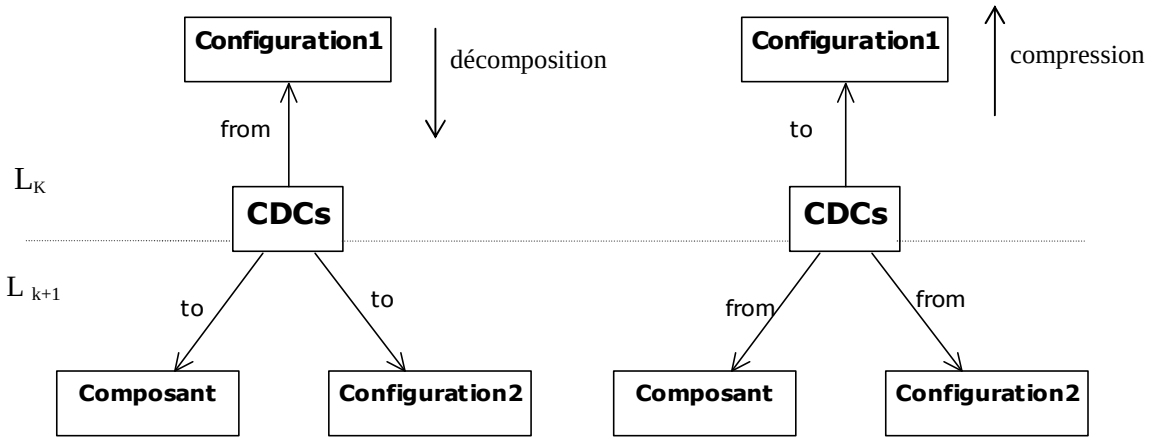


Figure 3.12- Liens possibles d'un connecteur de type CDCs.

La Figure 3.13 illustre un connecteur de type CDCs (*myCDC*) utilisé pour relier la configuration client-serveur (*CS-config)* définie au niveau hiérarchique (L_0) avec ses éléments internes à savoir les composants *Client* et *Serveur* définis au niveau hiérarchique inférieur (L_1). Par conséquent, l'interface du connecteur *myCDC* est spécifiée comme suit:

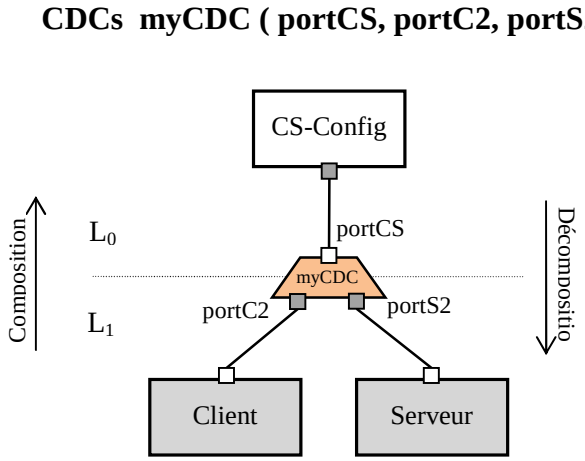


Figure 3.13- Le connecteur *myCDC* dans l'architecture client-serveur.

Où *portC2* et *portS2* représentent respectivement les ports du connecteur *myCDC* avec le composant *client* et le composant *Serveur*, et *portCS* représente le port du connecteur *myCDC* avec la configuration *CS-config*.

3.2.2.1.2 Connecteur de connexion structurelle (CCs)

L'assemblage est une technique fondamentale dans la perception d'un système comme un ensemble de composants interconnectés. De ce fait, l'assemblage est une relation permettant à un composant de référencer un autre composant pour utiliser « ses services ». Il ne doit pas être confondu avec une relation de composition, car l'assemblage concerne uniquement des éléments qui se trouvent au même niveau hiérarchique.

Nous proposons, dans C3, un connecteur de type CCs (*Connecteur de connexion structurelle*) utilisé pour connecter « *assembler* » des composants et/ou des configurations appartenant au même niveau de décomposition hiérarchique. Ce type de connecteur relie uniquement les ports *requis* aux ports *fournis*. A travers ces ports, les composants peuvent échanger des services entre eux. La forme générale de la signature des connecteurs de type CCs est :

CCs myCC_Connector ({X_i.requiredPort}, {Y_j.providedPort});

Avec X_i , Y_j $\in$ {composant, configuration},
X_i , Y_i $\in$ L_k // niveau(X_i) = niveau(Y_j) , le même niveau hiérarchique (L_k),
i = 1..m, j = 1..n, k = 0..r.
le nombre de ports de ce connecteur est (m+n)

Où (m + n) représente le nombre maximum d'éléments qui peuvent être reliés par un connecteur CC. (m+n) est le nombre de ports de *myConnector*. La correspondance entre les ports d'entrées et les ports de sorties est spécifiée par un protocole d'échange appelé « *glu* »

définie à l'intérieur du connecteur. Les différentes possibilités de liens, qu'un connecteur de connexion (*CCs*) peut avoir, sont illustrées par la Figure 3.14.

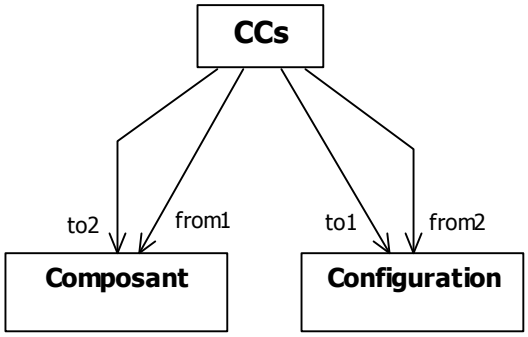


Figure 3.14- Liens possibles d'un connecteur de type CCs.

Concernant l'architecture client-serveur présentée précédemment, les composants *Client* et *Serveur* sont assemblés via le connecteur *myCC* de type *CCs* (cf. Figure 3.15). Par conséquent, le connecteur *myCC* définit deux ports ; le port *portC1* de type fourni et le port *portS1* de type requis. D'autre part le composant client devra avoir un port requis du même type que le port *portC1* et le composant serveur devra avoir aussi un port fourni du même type que le port *portS1*. La signature de *myCC* est spécifiée comme suit :

CCs myCC (portC1, portS1)

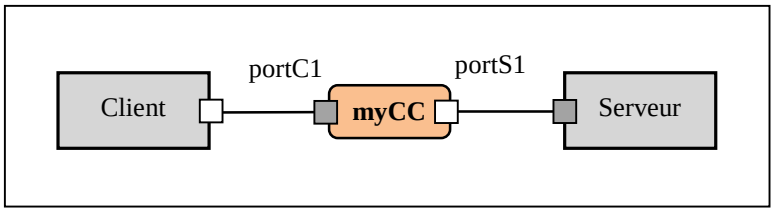


Figure 3.15- Le connecteur *myCC* dans l'architecture client-serveur.

3.2.2.1.3 Connecteur d'expansion/compression (ECC[3])

Dans la hiérarchie structurelle, nous avons besoin d'un autre type de connecteurs que nous utilisons pour mettre en interaction les interfaces des éléments qui appartiennent à différents niveaux d'abstraction. Dans cette perspective architecturale, nous avons analysé les interactions qui peuvent exister entre une configuration et ses éléments internes. Si une configuration est définie dans un niveau d'abstraction (L_i) alors ses éléments internes sont dans les niveaux (L_j) où ($j > i \geq 0$).

[3] *ECC : Expansion Compression Connector*

Nous avons constaté que les entrées d'une configuration peuvent être expansées vers les éléments internes. Par conséquent, les données acheminées vers les éléments internes ne sont pas forcement les mêmes données reçues par la configuration et inversement ; les sorties des éléments internes ne sont pas forcement les sorties de la configuration (exemples - changement de format de données en entrée et en sortie ; en entrée les données sont expansées et en sortie les données sont compressées, etc.) pour gérer ce type d'interaction, nous proposons un troisième type de connecteur d'expansion/compression.

Celui-ci est utilisé pour établir des liens de services entre une configuration et ses éléments internes. Il peut être utilisé en tant que opérateur d'expansion d'une fonction en plusieurs sous-fonctions et inversement (compression ou fusion de plusieurs sous-fonctions internes, pour fournir une seule fonction globale). Un connecteur ECC peut avoir une interface pour l'expansion et/ou une autre pour la compression. Son interface est, dans ce cas, définie suivant les deux signatures suivantes :

ECC myECC_expension (X.requiredPort , { Y_i.providedPort }) ; // interface d'expansion

ECC myECC_compression ({ Y_i.requiredPort } , X.providedPort) ; // interface de compression

où X est une configuration,
$Y \in$ {composant, configuration},
$i = 1,2,..,n$, et $n \leq$ nombre d'éléments internes de X ,
$X \in L_k$ et $Y_i \in L_{k+1}$; (L_k est le niveau hiérarchique numéro K).
Le nombre total de port de ce connecteur est $n+1$

Un connecteur de type ECC peut être implémenté soit avec une seule glu pour réaliser la fonction d'expansion ou celle de compression, soit il peut être implémenté en utilisant deux glus séparées, une pour la fonction d'expansion et l'autre pour la fonction de compression. La Figure 3.16 représente les différentes possibilités de connexions qu'un connecteur ECC peut avoir dans une architecture donnée.

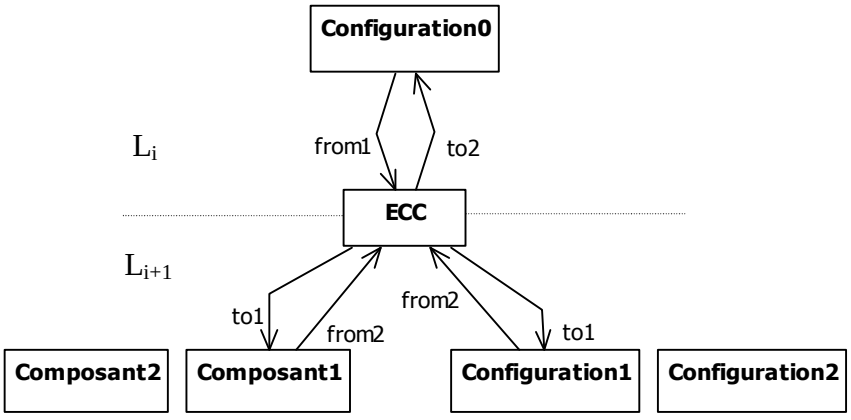


Figure 3.16- Liens possibles d'un connecteur de type ECC.

Remarque : les liens d'expansion et de compression ne s'appliquent pas forcément à l'ensemble des éléments internes d'une configuration. C'est le cas dans la Figure 3.16 où les éléments « composant2 » et « configuration2 » ne sont pas liés avec le connecteur ECC.

La Figure 3.17 illustre le connecteur *myECC* utilisé pour faire l'échange d'informations entre la configuration du serveur (*S-config*) et son composant interne coordinateur (*Coor*). Cependant, pour réaliser une communication bidirectionnelle entre le serveur et le composant coordinateur, *myECC* doit définir les ports suivants :

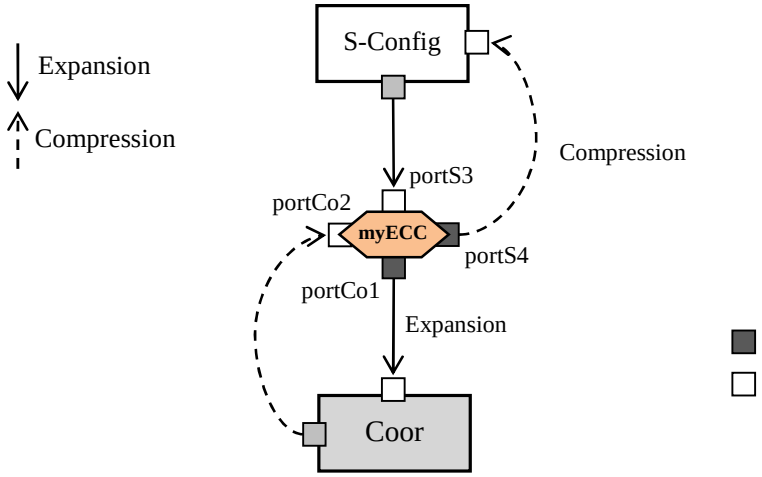


Figure 3.17- Le connecteur *myECC* dans l'architecture client-serveur.

portS3 comme port requis et *portCo1* comme port fourni, ces deux ports sont utilisés pour assurer la fonction d'expansion à partir de (*S-config*) au composant (*Coor*). *portCo2* comme port requis et *portS4* comme port fourni ; ces deux ports sont utilisés pour assurer la fonction de compression du composant coordinateur vers la configuration (*S-config*). Par conséquent, l'interface du connecteur *myECC* est définie comme suit:

ECC myECC (portS3, portCo1, portS4, portCo2)

Dans la hiérarchie structurelle d'une architecture, les éléments architecturaux sont connectés via leurs interfaces. Les connexions sont établies uniquement entre les ports syntaxiquement compatibles (ports de même type). De ce fait, la consistance des éléments assemblés, dans une hiérarchie de description structurelle, est vérifiée syntaxiquement.

3.2.2.2 Hiérarchie fonctionnelle (HF)

La hiérarchie fonctionnelle spécifie les fonctions et les caractéristiques d'entrées/sorties d'un système de type « boite noire » ; la connaissance et la réalisation des fonctions ne doivent pas apparaitre dans ce type de hiérarchie. Chaque élément structurel primitif a sa propre fonction. La fonction associée à l'élément se trouvant au plus haut niveau de la hiérarchie représente la fonction globale d'un système donné. L'architecture du système à ce niveau est perçue comme une boîte noire avec un ensemble de données en entrée (services

requis) et un ensemble de résultats en sortie (services fournis). Dans les niveaux inférieurs, chaque composant ou configuration possède sa propre fonction (*glu* pour les connecteurs). La fonction de chaque élément architectural peut être décrite par un diagramme d'état de transition, les réseaux de Petri ou un autre formalisme de spécification du comportement.

Une fonction représente la spécification du rôle qu'un élément peut jouer dans une architecture. Le rôle est défini par les relations entre les états possibles d'un élément conçu pour produire des résultats cohérents. La Figure 3.18 illustre la manière de décomposer une fonction F_0 au niveau (L_0) en un ensemble de sous-fonctions {F_{01}, F_{02}, F_{03}} au niveau (L_1). De la même manière, la fonction {F_{02}} se trouvant niveau (L_1) est décomposée en deux autres sous-fonctions {F_{021}, F_{022}} dans le niveau (L_2). Les fonctions feuilles de la hiérarchie représentent des fonctions primitives (non décomposables) associées aux composants primitifs disponibles dans la bibliothèque de l'architecte ou à implémenter par le développeur. L'ensemble des niveaux de décomposition représente la hiérarchie fonctionnelle du système.

Pour expliciter les liens qui peuvent exister entre les fonctions des éléments de la hiérarchie fonctionnelle, nous définirons dans les sous-sections suivantes trois types de connecteurs.

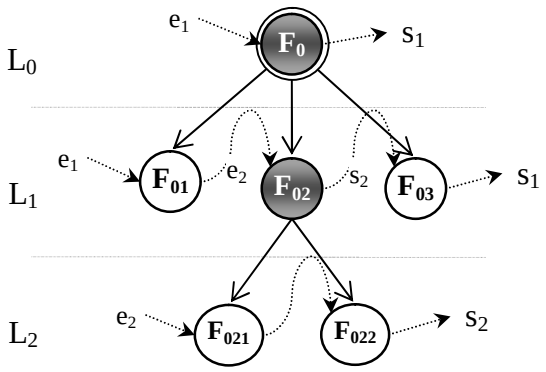


Figure 3.18- Hiérarchie fonctionnelle.

3.2.2.2.1 Connecteur de décomposition / composition fonctionnelle (CDCf)

Dans la hiérarchie fonctionnelle nous utilisons le connecteur CDC*f* pour relier chaque fonction à ses éventuelles sous-fonctions. Ce type de connecteur assure la communication entre une fonction donnée et ses sous-fonctions existantes à des différents niveaux d'abstraction. Ainsi, les connecteurs CDC*f* permettent de décrire l'arbre des fonctions associées aux éléments de l'architecture d'un système. La Figure 3.19 représente la notation adoptée pour ce type de connecteur et les connexions entre la fonction {F_0} de la Figure 3.18 et ses sous-fonctions {F_{01}, F_{02}, F_{03}}. La spécification de la signature de CDC*f* est identique à celle du CDC*s*.

CDC*f* myCDC*f* (portF$_0$, portF$_{01}$, portF$_{02}$, portF$_{03}$)

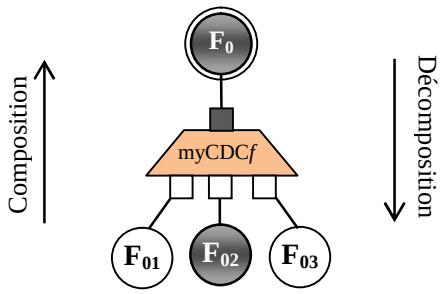


Figure 3.19- Les liens du connecteur *myCDCf*.

3.2.2.2.2 Connecteur de connexion fonctionnelle (CCf)

Le connecteur CC*f* est utilisé dans la hiérarchie fonctionnelle pour établir des connexions entre les entrées et les sorties des fonctions appartenant au même niveau hiérarchique. Si nous utilisons un diagramme d'état de transition pour spécifier le comportement des fonctions alors, la connexion entre deux fonctions sera réalisée par l'ajout d'une transition entre l'état terminal du premier diagramme et l'état initial du deuxième. Les entrées et sorties de chaque fonction représentent respectivement les données en entrée (*services requis*) et les résultats en sortie (*services fournis*). La Figure 3.20 illustre l'utilisation de deux instances (CC*f*1, CC*f*2) de type CC*f* pour relier la fonction (F_{02}) respectivement aux fonctions (F_{01}) et (F_{03}). De même la spécification de la signature de CC*f* est identique à celle de CC*s*.

3.2.2.2.3 Connecteur de lien d'identité (BIC)

Le connecteur de lien d'identité (*Binding Identity Connector*) est utilisé pour conserver l'identité et la traçabilité des entrées et des sorties des fonctions. Contrairement à la hiérarchie structurelle, dans la hiérarchie fonctionnelle, nous n'avons ni expansion des entrées ni compression des sorties lorsqu'on passe d'un niveau hiérarchique vers le niveau suivant. Pour cela, nous utilisons les mêmes mécanismes à tous les niveaux hiérarchiques. Ainsi, lorsque nous passons d'un niveau d'abstraction donné au niveau inférieur, nous avons toujours notre fonction mais décomposée en un ensemble de sous-fonctions afin de comprendre et d'analyser la complexité du comportement global du système. La spécification de la signature d'un connecteur BIC est la suivante.

// Lien d'identité entre deux entrées ou deux sorties associées à deux fonctions F_i et F_j
BIC myBIC (F_i.typePort , F_j.typePort);
Où F_i et F_j sont deux fonctions,
$F_i \in L_k$ et $F_j \in L_{k+1}$
L_k est le niveau d'abstraction ; k est un entier positif
typePort ∈ {required, provided}

La Figure 3.20 illustre la décomposition de la fonction (F_0) en trois sous-fonctions (F_{01}), (F_{02}) et (F_{03}). Nous avons (F_0) et (F_{01}) qui ont la même entrée (e_1) d'où le connecteur BIC1

qui relie les entrées des deux fonctions. De même, nous avons (F_0) et (F_{03}) qui ont la même sortie (s_1) d'où le deuxième connecteur BIC2 qui relie les sorties des deux fonctions.

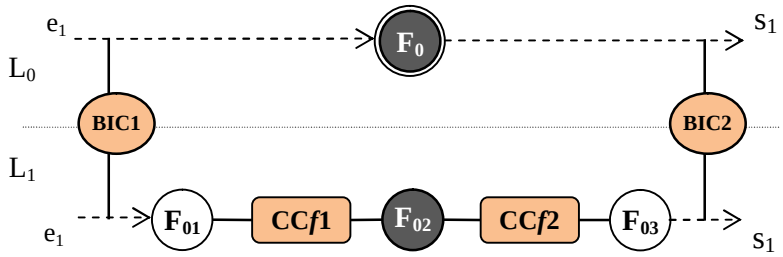


Figure 3.20- Illustration des connecteurs CC*f* et BIC.

Les signatures des connecteurs BIC1, BIC2, CC*f*1 et CC*f*2 sont définies comme suit :

BIC BIC1(portEntréeF$_0$, portEntréeF$_{01}$)
BIC BIC2(portSortieF$_0$, portSortieF$_{03}$)
CC*f* CC*f*1(requiredPortF$_{01}$, providedPortF$_{02}$)
CC*f* CC*f*2(requiredPortF$_{02}$, providedPortF$_{03}$)

La correction syntaxique, évoquée au niveau de la hiérarchie structurelle des éléments assemblés, n'assure pas la validité de l'architecture produite pour un système donné. Elle n'assure que la compatibilité des types d'interfaces des éléments assemblés. Bien que cette compatibilité d'interfaces soit nécessaire pour assurer l'échange d'informations entre éléments architecturaux, elle n'est pas suffisante pour vérifier leur collaboration, c'est-à-dire, leur sémantique de connexion nécessaire afin de produire des résultats cohérents. Donc, lors de l'élaboration de la hiérarchie fonctionnelle nous devons nous assurer de la compatibilité des fonctions associées aux éléments connectés à n'importe quel niveau de la hiérarchie.

Il est important de noter qu'il n'y a aucune relation de priorité entre les hiérarchies structurelle et fonctionnelle : le concepteur est libre de commencer par la hiérarchie qui lui convient en fonction de la nature des informations dont il dispose pour sa modélisation. Mais généralement, si la structure est connue, il est recommandé de commencer par la hiérarchie structurelle et à chaque niveau de celle-ci, on développe la hiérarchie fonctionnelle correspondante.

3.2.2.3 Hiérarchie conceptuelle (HC)

Il est impératif que la description et la manipulation de modèles complexes ne puissent être envisagées qu'à travers la notion de bibliothèque. Les éléments de cette dernière sont organisés via des hiérarchies conceptuelles de spécialisation. La hiérarchie conceptuelle permet à l'architecte de modéliser les relations entre les éléments de la même famille comme le montre la Figure 3.21. Dans cette hiérarchie les entités architecturales sont représentées par leurs types. Dont chacun est un élément de la bibliothèque et possède ses propres sous-éléments dans la même bibliothèque.

Ainsi, nous pouvons dresser une représentation graphique de la hiérarchie des entités de la même famille. Chaque famille, représentée par un graphe, a son propre nombre de niveaux hiérarchiques (*niveaux des sous types*). Au niveau le plus élevé de la hiérarchie, nous avons les types de base développés pour être réutilisés. Les types des niveaux intermédiaires sont créés en réutilisant les éléments des niveaux supérieurs pour produire d'autres sous types (*développement par la réutilisation et pour la réutilisation*). Les éléments du dernier niveau de la hiérarchie sont créés uniquement pour être utilisés dans la description d'une architecture. Les changements dans les bibliothèques sont possibles avec le respect de l'application des règles suivantes :

1- un nouvel élément peut toujours être ajouté à la bibliothèque,

2- un élément existant peut être augmenté par une nouvelle interface (requise ou fournie), mais une interface existante ne peut être supprimée.

Le respect de ces règles garantit la cohérence entre les éléments disponibles au niveau de la bibliothèque et les architectures conçues à partir de ces bibliothèques. Le mécanisme d'héritage associé à la relation de spécialisation est inspiré du mécanisme d'héritage des classes dans le paradigme objet. Ce mécanisme peut être utilisé pour définir des types d'éléments architecturaux concrets en tant que sous-éléments d'éléments abstraits en fournissant la structure et le comportement manquants.

Dans cette hiérarchie nous utilisons un mécanisme de spécialisation (exemple de sous-typage) comme outil de connexion entre un type et ses sous-types de la même bibliothèque. Cependant, le concepteur développe et classifie les éléments de la bibliothèque en fonction des besoins de développement d'architectures dans chaque domaine d'application. Le nombre de niveaux des sous-types dans une bibliothèque n'est pas limité. Néanmoins nous devons toujours rester à des niveaux raisonnables de spécialisation afin de maintenir un compromis entre l'*utilisation* et la *réutilisation* des éléments architecturaux. Pour mettre en œuvre cette hiérarchie nous définissons le type de connecteur suivant.

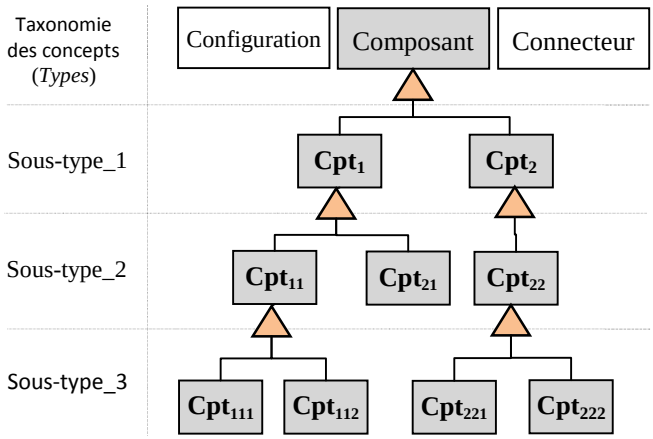


Figure 3.21- Hiérarchie conceptuelle d'un ensemble de composants.

3.2.2.3.1 Connecteur de spécialisation/généralisation (SGC[4])

Nous utilisons le connecteur spécialisation/généralisation afin de présenter les liens entre les types d'éléments provenant du même type. Parmi les implémentations courantes de ce mécanisme de connexion, nous avons l'héritage dans les langages orientés objet. Donc, nous pouvons construire facilement des arbres représentant tous les types de classification d'une bibliothèque d'éléments architecturaux. La Figure 3.22 représente la notation adoptée pour ce type de connecteur (SGC).

Le connecteur proposé dans C3 pour modéliser la spécialisation/généralisation sera utilisé pour créer des hiérarchies conceptuelles spécialisées de type d'éléments architecturaux. Les types abstraits pourront être utilisés comme un moyen simple pour factoriser un ensemble de types partageant des propriétés et/ou des opérations communes. La spécification de la signature d'un connecteur SGC est la suivante.

// spécialisation d'un type élément X en plusieurs sous-types plus spécialisés

SGC mySGC (X.typePort , { Y_i .sous_typePort});

Où X est un type d'élément,
$Y_i \in$ {sous type de X}, i=*1..n*
$X \in L_k$ et $Y_i \in L_{k+1}$
L_k est le niveau d'abstraction ; k est un entier positif

L'exemple de la Figure 3.22 illustre la spécialisation d'un type de composant abstrait (*clientComponent*) en deux sous-composants (*clientSyn* et *clientAsyn*). Chaque sous-composant fournit un mode de communication bien particulier ; le composant *clientSyn* propose le mode synchrone pour la communication tandis que le composant *clientAsyn* propose le mode de communication asynchrone. La signature globale du connecteur sera définie comme suit :

SGC mySGC (typePort, s_typePort, as_typePort)

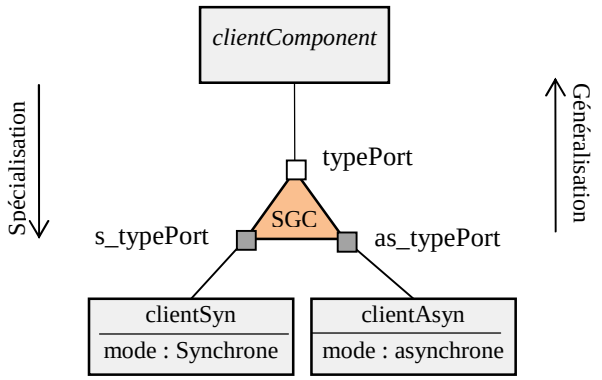


Figure 3.22- Spécialisation d'un type de composant par un connecteur SGC.

[4] *SGC : Specialisation Generalisation Connector*

3.2.2.4 Hiérarchie de méta-modélisation (HM)

La hiérarchie de méta-modélisation est une pyramide composée exactement de quatre niveaux d'architectures. Au niveau le plus bas, la strate A0 correspond au système réel (*application instance*). Un modèle d'architecture *représente* ce système au niveau A1. Ce modèle est *conforme* à son méta-modèle défini au niveau A2 et le méta-modèle lui-même est *conforme* au méta-méta-modèle au niveau A3. Le méta-méta-modèle est *conforme* à lui-même. [OMG 2003a] [OMG 2007]. La Figure 3.23 illustre ces quatre niveaux ainsi que les relations eux.

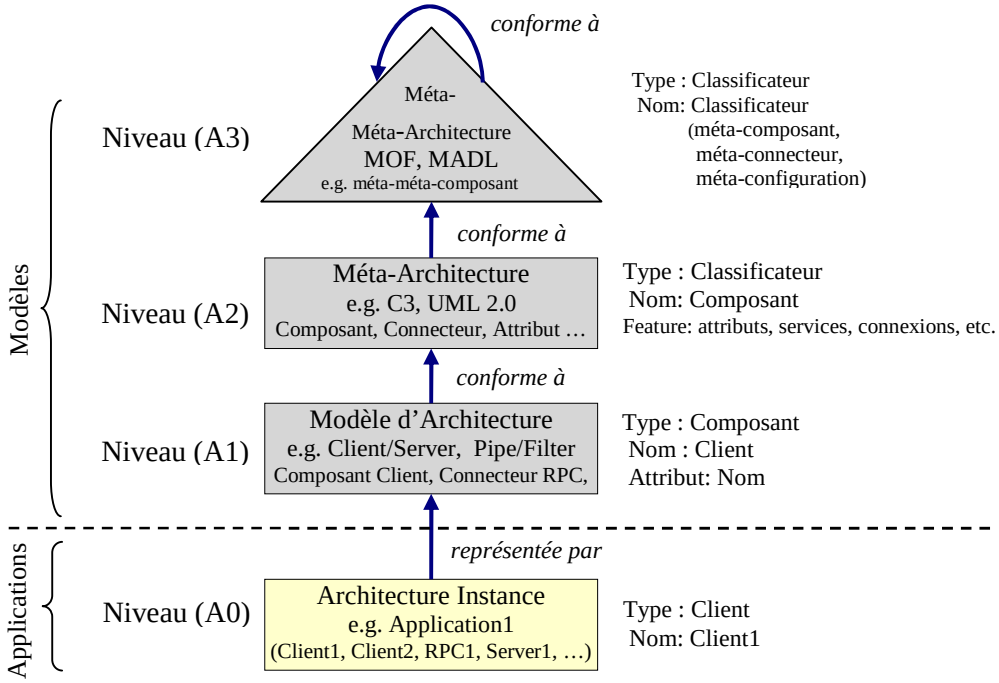


Figure 3.23 - Hiérarchie de méta-modélisation.

Niveau A0 : est le niveau qui représente le monde réel instancié à partir du niveau modèle d'architecture (A1). A ce niveau, le développeur a la possibilité de sélectionner et d'instancier autant de fois les éléments nécessaires pour décrire son application. Les instances sont créées à partir des types d'éléments qui sont définis au niveau A1. Ces instances sont assemblées en respectant les différentes contraintes définies aussi au niveau (A1).

Niveau A1 : ce niveau est appelé niveau modèle d'architecture. A ce niveau, nous avons des modèles d'éléments architecturaux décrits en utilisant des langages de description d'architectures ou des langages de notations définis au niveau (A2) (exemple - UML 2.0).

Niveau A2 : est le niveau méta-architecture définit le langage de description ou de notation utilisé pour décrire les modèles d'architectures au niveau (A1). Toutefois les opérations menées à ce niveau sont toujours en conformité avec le niveau supérieur de la pyramide.

Niveau A3 : c'est le dernier niveau de la pyramide, appelé aussi niveau méta-méta-architecture. A ce niveau, on trouve les concepts les plus généraux utilisés pour définir de nouveaux langages de description d'architecture. Dans des travaux antérieurs de notre équipe de recherche, nous avons défini notre propre méta-méta-modèle d'architecture appelé MADL [Smeda *et al.* 2005] (cf. Annexe B). Le méta-modèle C3 est défini en conformité avec MADL. La vision de conception de MADL est proche de MOF mais plus orientée composant. Le modèle MADL est orienté composant dans le sens où tout est composant (tous les éléments sont des sous-types du type abstrait *Composant*, comme dans le modèle UML où tout est sous-classe de la classe abstraite *modelElement*). Nous considérons que le composant est l'entité architecturale de base du modèle. MADL est un noyau réflexif dédié à la méta-méta-architecture. Il définit les concepts et les relations de base comme le méta-composant, le méta-connecteur, la méta-architecture et la méta-interface.

Pour relier chaque élément architectural avec son type dans le niveau supérieur de la hiérarchie de méta-modélisation nous proposons le type de connecteur suivant :

3.2.2.4.1 Connecteur d'instanciation (IOC[5])

Le connecteur IOC est utilisé pour établir la connexion entre les éléments d'un niveau donné (*instances*) avec leurs types définis dans le niveau supérieur (*modèles*) dans la hiérarchie de méta-modélisation. La spécification de la signature d'un connecteur IOC est définie comme suit :

// instanciation d'un type élément X en plusieurs instances Y_i
IOC myIOC (X.typePort , { Yi .instancePort});

Où X est un type d'élément architectural,
$Y_i \in$ {instance direct de X}, i=*1..n*
$X \in L_k$ et $Y_i \in L_{k-1}$,
L_k est le niveau de méta-modélisation (K=1..3).

L'exemple de la Figure 3.24 montre deux instanciations du même type de composant *clientComponent* au niveau (A1) sur des composants particuliers (*instances*) au niveau (A_0) d'une architecture logicielle. Dans le premier cas, l'instanciation crée un client avec un seul port (*p1*) de type « *request*» ; dans le second cas, l'instanciation crée un composant client avec deux ports (*p1* et *p2*) de type « *request*». La syntaxe de la signature de ce connecteur est :

IOC myIOC (typePort, instPort1, instPort2)

[5] *IOC* : *Instance Of Connector*

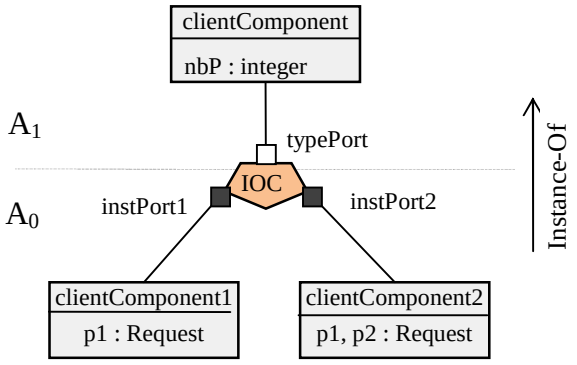


Figure 3.24- Connecteur d'instanciation entre un type de composant et ses deux instances.

3.3 Architecture physique (AP)

Cette section présente deux nouveaux concepts de la description d'une architecture logicielle dans C3 qui viennent compléter la description du modèle type d'architecture présentée dans les sections précédentes. Le premier est le modèle logique. Il permet de décrire une architecture logicielle typée constituée d'un graphe d'instances de composants et de configurations interconnectées par des instances de connecteurs. Le deuxième modèle est le modèle physique. C'est une représentation concrète de la topologie de l'application qui représente une image interne de l'architecture logique. Cette image est construite sous la forme d'un graphe dont les nœuds sont des instances d'un gestionnaire de connexions. Chaque instance créée correspond à un composant ou à une configuration instanciée pour construire l'application réelle. Les nœuds de ce graphe sont reliés par des arcs de trois types dont chacun correspond à un type particulier de connecteur. L'architecture physique est conçue pour servir de support aux opérations de mise à jour et d'évolution de l'application instance comme l'ajout, la suppression et le remplacement des éléments de l'application.

3.3.1 Gestionnaire de connexions

L'architecture physique est décrite en utilisant seulement deux niveaux d'abstractions à savoir le niveau type et le niveau instance comme le montre la Figure 3.25. Au niveau type nous avons la définition d'un gestionnaire de connexions (CM pour *Connection Manager*) représenté par une classe qui encapsule les différentes informations (cf. Figure 3.26) sur les connecteurs qui peuvent être reliés à un composant ou une configuration dans une application donnée.

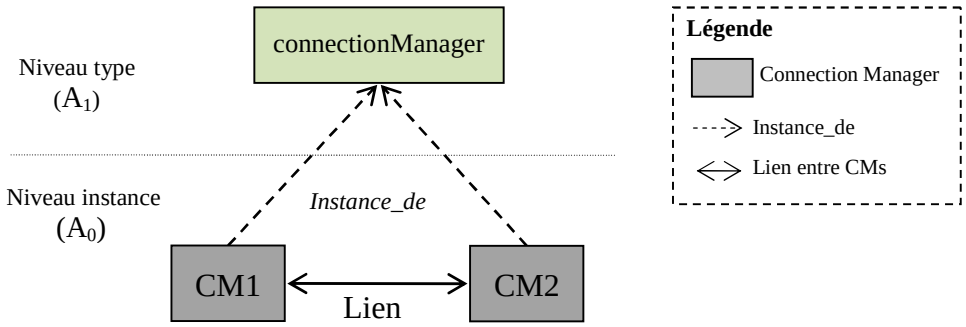


Figure 3.25- Niveau d'abstraction dans l'architecture physique.

```
ConnectionManager Name {
    String ElementName ;
    List CC_Links ;
    String CDC_Link ;
    String ECC_Link ;
}
```

Figure 3.26- Structure du gestionnaire de connexions.

Chaque CM est identifié par un nom et possède quatre attributs.

1. *ElementName* : représente le nom de l'élément architectural associé à ce CM, c'est-à-dire le nom du composant ou de la configuration correspondante au niveau de l'architecture logique (c'est le principe même d'un gestionnaire de connexions);

2. *CC_Links* : liste de noms des connecteurs de connexions (CC) reliés à l'élément architectural associé à ce CM (un composant ou une configuration peuvent être reliés à plusieurs CC) ;

3. *CDC_link* : le nom du connecteur de décomposition composition (CDC) relié à l'élément architectural associé à ce CM. Notons qu'une configuration ne peut avoir qu'un seul connecteur de décomposition structurelle;

4. *ECC_Link* : le nom du connecteur d'expansion compression (ECC) relié à l'élément architectural associé à ce CM. Dans une hiérarchie structurelle, une configuration peut avoir plusieurs ECC relié avec elle en entrée et en sortie.

3.3.2 Operations possibles sur un gestionnaire de connexions

Les opérations possibles que nous pouvons appliquer sur le gestionnaire de connexions sont les suivantes:

1. *Instanciation() :* le gestionnaire de connexions est instancié au niveau application (A_0) de l'architecture physique. Chaque fois qu'un élément architectural (composant ou configuration) est instancié au niveau de l'application un CM associé est automatiquement créé dans l'architecture physique.

2. *Installation() :* chaque fois qu'un connecteur est installé au niveau de l'application entre un ensemble d'instances d'éléments architecturaux, les attributs des CMs associés sont mis à jour avec les informations nécessaires relatives à cette instance de connecteur.

3. *Propagation() :* un mécanisme de propagation est nécessaire pour mettre à jour les informations sur les liens nécessaires entre les CMs. Ces liens sont définis par l'interface du connecteur installé au niveau de l'application.

Pour illustrer la description de l'architecture physique, nous reprenons l'architecture logique de l'exemple client-serveur. L'application client-serveur illustrée par la Figure 3.27 représente une application instance de l'architecture logique client serveur. Dans cette application nous avons une configuration de type client-serveur (CS) composée à partir de deux composants de type clients (C1, C2) reliés à un composant de type serveur (S) par le biais de deux connecteurs de type CC (RPC1, RPC2).

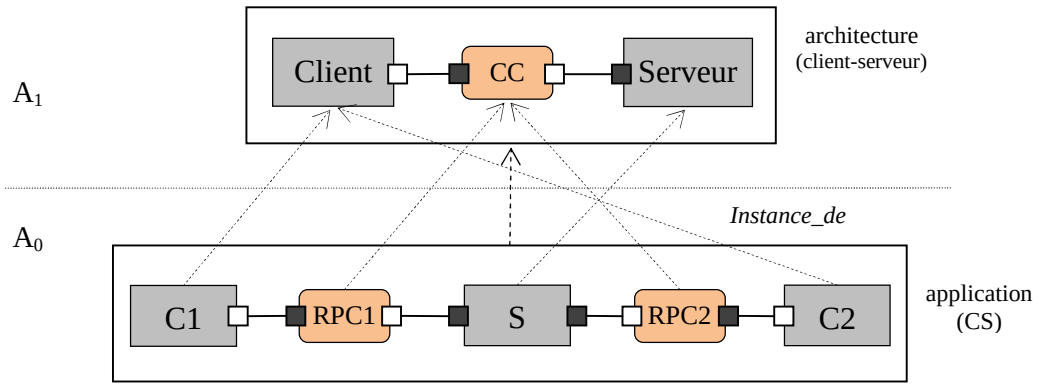


Figure 3.27- Architecture client-serveur et une application instance.

Le type de composant serveur est défini par une configuration qui renferme des composants internes de type (*coordinator*, *securityManager* et *dataBase*). Ceux-là sont assemblés en utilisant des connecteurs de type CC et sont reliés à leur configuration via des connecteurs de type CDC et ECC.

La Figure 3.28 illustre l'architecture physique correspondante à l'application instance (CS) définie dans la Figure 3.27. Cette architecture physique est représentée par un graphe dont les nœuds sont des CMs et les arcs des liens de connecteurs.

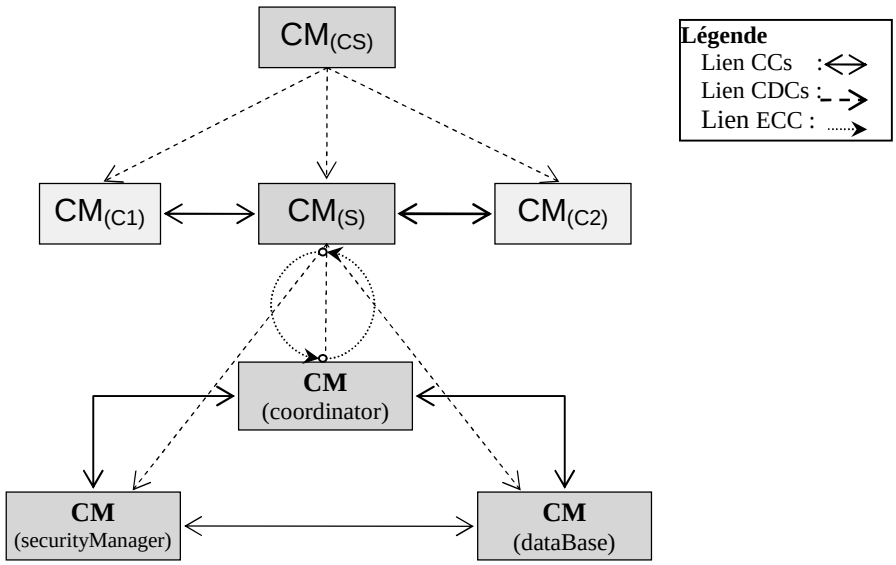


Figure 3.28- Architecture physique de l'application client-serveur.

Une fois que l'application est construite par le développeur, l'architecture physique correspondante est également construite en parallèle. Par la suite, si nous avons besoin d'intervenir sur l'application pour la mettre à jour ou la faire évoluer, nous devons repérer les éléments concernés sur l'architecture physique en utilisant des opérations de recherche et de mise à jour dans le graphe des CMs (exemple : *ajouter(noeud)*, *supprimer(noeud)* ou *remplacer(noeud)*).

Enfin, nous pouvons représenter la relation entre l'architecture logique (AL) et l'architecture physique (AP) par un modèle d'architecture décrit en C3, où les architectures logique et physique sont représentées par deux composants et la relation entre elles par un connecteur de connexion comme le montre la Figure 3.29. Dans cette figure l'architecture logique prend en charge les trois premiers niveaux de modélisation (A0, A1 et A2). Par contre l'architecture physique définit uniquement les deux premier niveaux (*type*, *instance*). L'AL est doté d'un port logique fourni (PL) et l'AP d'un port physique requis (PP).

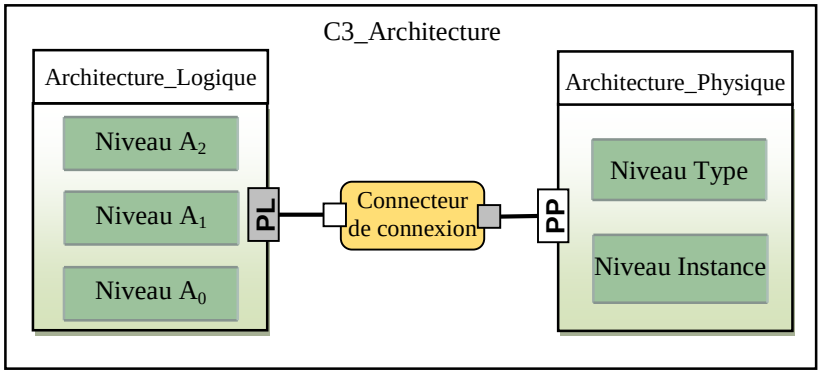


Figure 3.29- Représentation architecturale de la relation entre l'AL et l'AP.

Toute action réalisée sur l'architecture logique provoque l'envoi d'un message de celle-ci vers l'architecture physique. La réception de ce message par cette dernière provoque l'invocation et l'exécution d'une action locale.

Les opérations échangées entre les deux types d'architectures sont:

1. L'instanciation d'un composant ou d'une configuration au niveau de l'AL provoque l'envoi du message « *CM_Creation* » à l'AP. Ce message invoque une opération de création d'une instance du gestionnaire de connexions au niveau de l'AP.
2. L'instanciation d'un connecteur au niveau de l'AL provoque l'envoi du message « *CM_Connection* » à l'AP. Ce message déclenche une opération de création d'un ensemble déterminé de liens entre les instances du gestionnaire de connexions qui correspondent aux instances de composants et de configurations reliés par le connecteur instancié au niveau de l'architecture logique.
3. Chaque opération de mise à jour (remplacement ou suppression) d'un composant ou d'une configuration au niveau de l'AL provoque l'envoi du message « *CM_Update* » à l'AP. Celui-ci invoque une opération de mise à jour des liens de l'instance du gestionnaire de connexions correspondant à l'élément concerné par la mise à jour.
4. Chaque opération de mise à jour (remplacement ou suppression) d'un connecteur au niveau de l'AL provoque l'envoi du message « *Link_Update* » à l'AP. Celui-ci déclenche une opération de mise à jour des liens entre deux ou plusieurs instances de gestionnaire de connexion. Ces instances correspondent à tous les éléments reliés avec cette instance de connecteur.

Notons que tous les messages échangés entre l'AL et l'AP passent via leurs ports respectifs PL (*Port Logique*) et PP (*Port Physique*).

3.4 Bilan de C3

Dans ce chapitre, nous avons illustré les différents éléments du méta-modèle C3 pour spécifier la description d'une architecture logicielle. Nous avons également proposé un formalisme de représentation des concepts architecturaux à des niveaux conceptuels différents. Dans cette section, nous fournissons les éléments de synthèses relatifs à ce modèle de description d'architecture vis-à-vis des critères introduits dans le chapitre 2 de cette thèse.

3.4.1 Eléments de synthèse

L'idée clé de notre modèle de description d'architecture est de donner beaucoup plus d'importance aux concepts de connecteur et de configuration et de les rendre des éléments architecturaux de premières classes au même niveau conceptuel que les composants. Les connecteurs et les configurations dans C3 sont des éléments typés, disponibles dans des bibliothèques et utilisables comme les COTS. Nous nous sommes intéressés par la

modélisation des différents types d'interactions qui peuvent avoir lieu entre les composants d'une architecture. Ces interactions sont modélisées par un ensemble bien défini de connecteurs. La Figure 3.30 illustre la hiérarchie conceptuelle des connecteurs proposés.

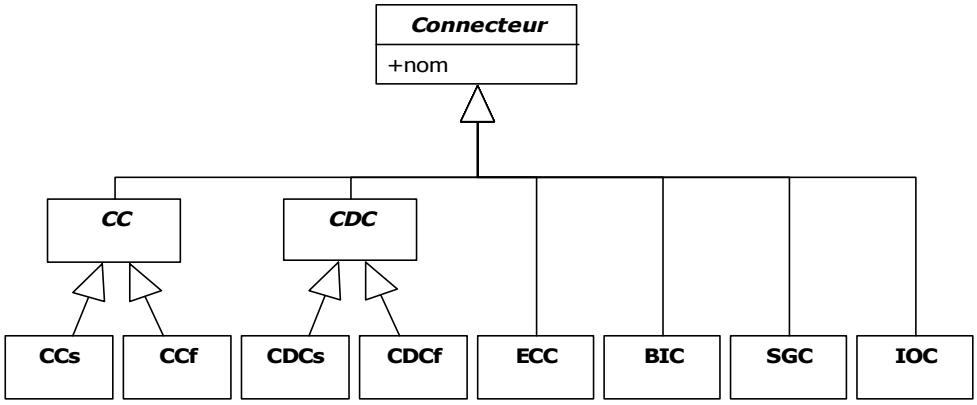


Figure 3.30 - Hiérarchie conceptuelle des connecteurs proposés.

3.4.2 Positionnement C3 par rapport aux autres ADLs

Les tableaux 3.1 et 3.2 récapitulent respectivement le positionnement des connecteurs et des configurations C3 par rapport aux critères d'évaluation des langages de description d'architectures présentés dans le chapitre 2.

C3	Propriétés des connecteurs dans C3				
	Interface	**Types**	**Contraintes**	**Evolution**	**PNF**
proposition d'une nouvelle structure explicite pour les connecteurs, qui incluent les liens d'attachements	interface avec chaque composant ou configuration via des ports ; les éléments d'interfaces sont de type requis / fourni	système de typage extensible, basé sur les protocoles	chaque port participe à un ou plusieurs liens entre composants	sous-typage structurelle via le mécanisme d'extension	possible mais sans aucun traitement explicite

Tableau 3.1- Evaluation des connecteurs C3 vis-à-vis des propriétés fixées dans le chapitre 2.

C3	Propriétés des configurations dans C3				
	Composition	**Raffinement et Traçabilité**	**Passage à l'échelle**	**Contraintes**	**Evolution**
configuration explicite	supporté via l'architecture des composants et connecteurs internes	supportés via les différents types de hiérarchies	aidé par les configurations explicites et le nombre variable de ports de connecteurs et de composants	les ports de connecteurs doivent être reliés uniquement aux ports de composants et de configurations compatibles	aidé par les configurations explicites ; peu d'interdépendance entre composants ; connecteurs hétérogènes

Tableau 3.2- Evaluation des configurations C3 vis-à-vis des propriétés fixées dans le chapitre 2.

Le méta-modèle C3 s'efforce d'offrir un bon niveau de réutilisation. En effet ; il favorise :

- *L'extensibilité :* chaque nouvel ajout d'un composant à une architecture ne perturbe aucunement la cohérence de l'architecture, du moment que l'opération d'ajout respecte la compatibilité de l'assemblage déjà présent.
- *La compositionalité :* la relation de composition fournie par C3 permet la spécification d'éléments composites dans une architecture via le concept de configuration.
- *La remplaçabilité :* un élément architectural peut être substitué à un autre du moment qu'il possède les même interfaces.
- *L'évolutivité :* la structure ou le comportement interne d'un élément architectural peut être modifié sans impact sur la façon dont les autres éléments l'utilisent, du moment qu'ils gardent la relation au niveau interface uniquement.

C3 offre un bon support pour le développement d'architectures évolutives, en se basant sur une description minimaliste des architectures logicielles. Le formalisme du modèle en Y contribue à la simplicité du méta-modèle C3.

3.5 Formalisme de représentation des concepts de C3

La spécification du méta-modèle C3 doit être aussi simple que possible, tant sur le plan des concepts manipulés que sur la quantité des architectures à décrire. Compte tenu des fonctionnalités offertes par C3, le formalisme choisi doit permettre de distinguer les différentes vues architecturales, ainsi que les éléments concernés. En outre, l'architecte doit pouvoir hiérarchiser les composants, les connecteurs et les configurations et disposer de plusieurs vues sur ses architectures.

Le modèle Y (MY) [Smeda *et al.* 2008] est un formalisme qui intègre naturellement ces fonctionnalités et qui a en outre été utilisé par notre équipe pour décrire les architectures logicielles à base de composants. En utilisant MY comme support, une architecture C3 peut être décrite par trois aspects correspondant aux trois branches de Y : composant, connecteur et configuration. Ces aspects représentent tout ce qui est lié à la description d'architectures.

3.5.1 MY : La méta-modélisation en Y

La modélisation, la méta-modélisation et les bibliothèques de composants réutilisables sont des techniques éprouvées. Elles sont utilisées dans l'ingénierie de la connaissance (*Knowledge Engineering*, KE). La modélisation et la méta-modélisation sont des techniques utilisées pour représenter des systèmes à un niveau élevé qui abstrait les considérations d'implémentation et se concentrent sur la conception [Menzies 2002]. Dans le domaine de la description d'architectures, on parle de la méta-architecture pour évoquer une architecture particulière d'un système logiciel.

Les bibliothèques de composants réutilisables ont été décrites dans différentes approches de KE [Motta 2000]. Après avoir examiné ces bibliothèques, nous avons constaté qu'elles font face à des problèmes très proches des nôtres. En ce qui nous concerne : comment décrire un composant, un connecteur ou une configuration, ce que devrait être le niveau de granularité d'un composant, d'un connecteur ou d'une configuration, comment structurer et classer les éléments de telle sorte que les utilisateurs puissent facilement comprendre, trouver, configurer et intégrer ces éléments afin de construire leurs architectures.

Les approches classiques de KE telles que CommonKADS [Breuker et Van de Velde 1994], TINA [Benjamins 1995], VITAL [Shabolt *et al.* 1993], Protégé [Angele *et al.* 1998], et MY [Oussalah 2002] décrivent des systèmes à base de connaissance (*SBC*) en utilisant trois composants complémentaires : tâche, méthode de résolution de problème, et domaine. La séparation des méthodes (utilisées pour réaliser une tâche donnée) de la tâche et du domaine permet de définir une application de SBC comme une combinaison de ces trois composants. L'approche MY combine la méta-modélisation et les bibliothèques de composants réutilisables.

3.5.2 Le Modèle MY

En utilisant le modèle MY, l'architecture logicielle d'un système peut être décrite par trois aspects : composant, connecteur et configuration.

- Le composant représente les unités de calculs du système,
- Le connecteur représente les interactions entre les composants du système,
- La configuration représente la topologie ou la structure du système.

Ces trois concepts sont interdépendants et nécessaires pour décrire une architecture logicielle. Ils sont définis dans un objectif de réutilisation pour décrire différents systèmes. Par conséquent, nous pouvons réutiliser un modèle pour décrire plusieurs systèmes appartenant à des domaines différents.

- Un composant peut être utilisé pour décrire un serveur, un client, une base de données, etc.
- Un connecteur peut être utilisé pour décrire un appel de fonction, un RPC, une pipe, etc.
- Une configuration peut être utilisée pour décrire différents styles d'un système, Client-Serveur, Pipe-Filter, etc.

Un système peut être bien décrit en utilisant un tel modèle, qui a la capacité de représenter tous les aspects d'une architecture logicielle.

3.5.2.1 Les concepts de base

Les trois concepts principaux de la description architecturale sont les composants, les connecteurs, et les configurations. Nous modélisons ces concepts en utilisant trois aspects : aspect composant, aspect connecteur et aspect configuration. Chaque branche de la lettre Y

décrit un concept architectural, comme le montre la Figure 3.31. Par exemple, l'aspect connecteur permet de décrire des connecteurs génériques. Le centre de l'architecture en Y décrit un connecteur primitif. Notons que plus le connecteur est loin du centre, plus il est réutilisable. De plus, afin de garder les descriptions des trois concepts (composant, connecteur, configuration) indépendantes les unes des autres, nous explicitons deux relations : relation inter-concepts et relation intra-concepts. La relation *inter* décrit le lien entre deux concepts de type différents, alors que la relation *intra* décrit le lien entre deux concepts du même type. Ces relations sont utilisées dans le MY comme un outil de composition et d'assemblage pour décrire des architectures logicielles.

La relation *intra* se décline sous les formes Composant-Composant, Connecteur-Connecteur et Configuration-Configuration. La spécialisation (*connecteur SGC*), la composition (*connecteur CDC*), l'expansion (*connecteur ECC*) et l'instanciation (*connecteur IOC*) définis dans C3 sont des relations intra-concepts. A titre d'exemple, la spécialisation d'une configuration est perçue comme la combinaison d'une spécialisation de ses composants et connecteurs.

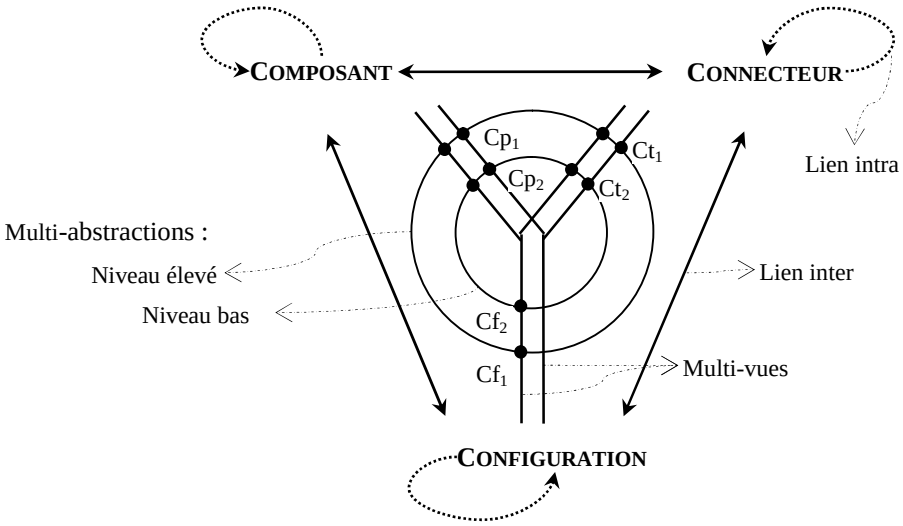


Figure 3.31 - La représentation des concepts architecturaux en utilisant le modèle en Y.

La relation *inter* se décline sous les formes Configuration-Composant, Configuration-Connecteur et Composant-Connecteur. Des relations explicites mise en œuvre à l'aide des connexions entre les ports des éléments architecturaux (*connecteur CC).* Pour assurer la cohérence de l'assemblage résultat, ces relations doivent respecter la compatibilité de type entre les ports correspondants.

3.5.2.2 Multi-abstractions et multi-vue

A fin de capturer tous les aspects des architectures logicielles et encapsuler leur complexité, MY décrit les architectures selon deux décompositions : multi-abstractions et multi-vue comme illustrées dans la Figure 3.31.

3.5.2.2.1 La décomposition multi-abstractions

La décomposition multi-abstractions définit différentes abstractions pour un élément donné. Aussi, une architecture peut être décrite en utilisant différents niveaux d'abstraction. Chaque niveau d'abstraction est représenté par un cercle dans le modèle en Y, à partir du niveau le plus élevé (cercle de fort diamètre) au niveau le plus bas (cercle de faible diamètre). Les deux appellations données aux cercles, correspondent à l'abstraction de décomposition et de composition structurelle ou fonctionnelle, à l'abstraction d'expansion/compression structurelle et à l'abstraction de spécialisation/généralisation de concepts. En règle générale, pour obtenir une architecture simplifiée avec un minimum d'éléments, nous devons logiquement la décrire à des niveaux d'abstractions plus élevés.

3.5.2.2.2 La décomposition multi-vue

La décomposition multi-vue permet la définition de différentes vues pour un système donné. Le fait d'avoir différentes vues d'un système clarifie ses différents aspects importants et aide à gérer sa complexité. Dans le méta-modèle C3 nous avons identifié les vues suivantes : la vue structurelle, la vue fonctionnelle, la vue conceptuelle et la vue méta-modélisation représentées dans C3 respectivement par les hiérarchies structurelle, fonctionnelles, conceptuelle et de méta-modélisation.

Le modèle MY représente différentes vues en utilisant plusieurs Ys, chaque Y représente une vue du système et comment les différents éléments architecturaux sont supportés dans cette vue (explicitement ou implicitement). Par exemple, dans le système donné en Figure 3.32, les composants, les connecteurs et les configurations sont explicitement représentés dans la vue structurelle et conceptuelle. Cependant, dans la vue fonctionnelle seuls les composants sont explicitement représentés, alors que les connecteurs et les configurations sont implicitement représentés. La vue de méta-modélisation représente le concept de composant à trois niveaux de modélisation (composant, composant type, méta-composant). Les niveaux d'abstractions dans chaque vue représentent la bibliothèque associée à chaque vue.

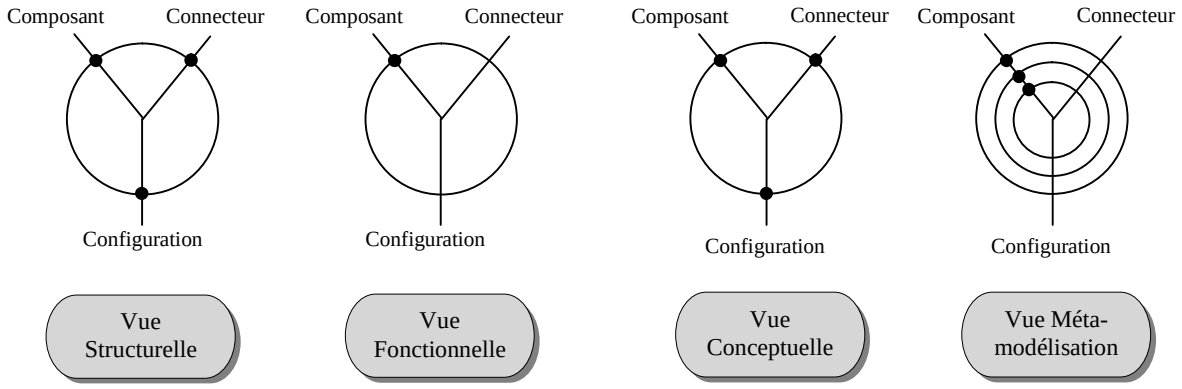


Figure 3.32- La représentation de vues multiples en utilisant MY.

3.5.2.3 Modélisation Multidimensionnelles de l'architecture logicielle

La modélisation de l'architecture logicielle permet de définir des architectures selon trois dimensions : niveau de conception, éléments architecturaux réutilisables, et processus de réutilisation, comme le montre la Figure 3.33.

3.5.2.3.1 Dimension niveau de conception

La dimension niveau de conception représente les différents niveaux de modélisation des éléments architecturaux (*vue de méta-modélisation*), à partir de la définition de la méta-méta-modélisation jusqu'au niveau application. Nous pouvons distinguer quatre niveaux conceptuels : méta-méta-Architecture (A3), méta-architecture (A2), architecture (A1), et application (A0).

Nous avons doté le modèle MY d'une bibliothèque d'éléments de différents niveaux de conception. La bibliothèque est multi-hiérarchique avec quatre niveaux différents : méta-méta-architecture pour stocker des éléments méta-méta tels que les méta-composants, les méta-connecteurs et les méta-configurations ; la bibliothèque de méta-architecture pour stocker des éléments méta tels que les composants, les connecteurs et les configurations ; la bibliothèque d'architecture pour stocker des types d'éléments d'architecture tels que les composants (serveur, client), les connecteurs (RPC) et les configurations (client-serveur) et la bibliothèque niveau application pour stocker les instances. L'utilisateur final peut choisir l'élément voulu, au degré de spécification voulu.

3.5.2.3.2 Dimension éléments réutilisables

Cette dimension permet d'identifier les éléments architecturaux réutilisables pour chaque niveau conceptuel. Ces éléments sont alors stockés dans une bibliothèque liée à chaque niveau conceptuel et utilisée par son niveau inférieur. Par exemple, les éléments réutilisables du niveau méta architecture sont des composants, des connecteurs, des configurations et des ports. Ils sont stockés dans la bibliothèque (niveau A2) et utilisés au niveau architecture.

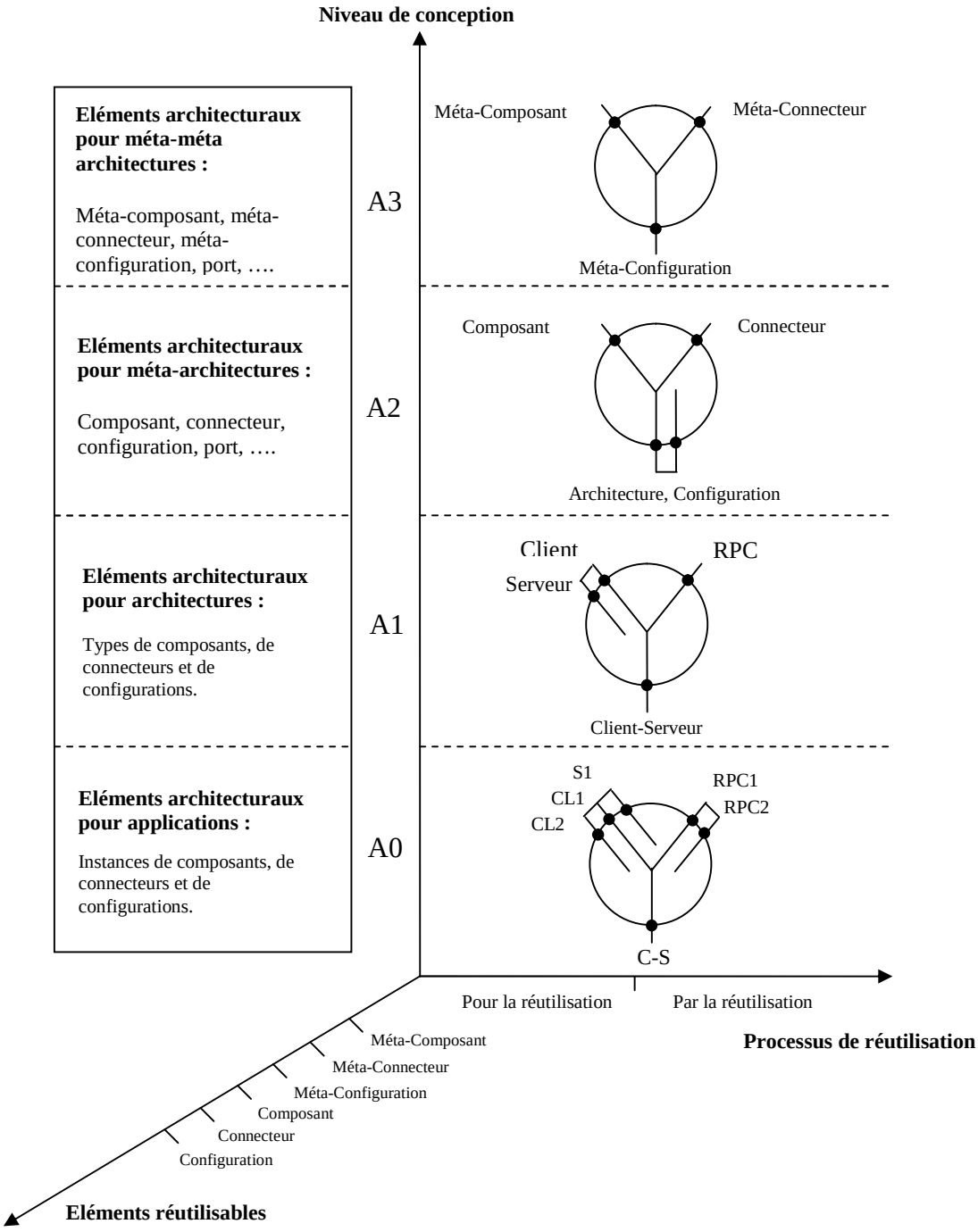


Figure 3.33- Les trois dimensions de l’architecture logicielle.

Nous avons identifié quatre catégories d'utilisateurs pour l'architecture en Y. Chacune est associée à un niveau d'architecture donné (A3, A2, A1 et A0). En plus, une cinquième catégorie d'utilisateur est nécessaire pour gérer la bibliothèque. Ainsi, nous avons :

1. CI (*Constructeur d'Infrastructure*) : définit les concepts de base architecturaux sur lesquels le Constructeur du Langage d'Architecture (CA) se base pour établir un langage de description d'architecture. Le CI est concerné par le niveau conceptuel A3.
2. CA (*Constructeur du langage d'Architecture*) : définit l'ADL nécessaire par l'instanciation des concepts fournis par le CI. Le CA est concerné par le niveau conceptuel A2.
3. AA (*Architecte d'Applications*) : son rôle est de concevoir et décrire l'architecture qui sera utilisée pour le développement de l'application (décrit l'architecture de l'application). Le AA est concerné par le niveau conceptuel A1.
4. DA (*Développeur d'Applications*) : son rôle est de développer une application en instanciant le niveau A1. Dans certains cas, on peut automatiquement générer le code à partir de l'architecture, ainsi le DA n'intervient pratiquement pas. Le DA est concerné par le niveau A0.
5. IR (*Ingénieur de la Réutilisation*) : construit et gère toutes les bibliothèques, qui sont hiérarchisées suivant les quatre niveaux conceptuels.

3.5.2.3.3 Dimension processus de réutilisation

Cette troisième dimension permet aux éléments architecturaux d'être vus selon leur construction : (*pour la réutilisation*) afin de construire la bibliothèque, et leur utilisation en les choisissant à partir de la bibliothèque (*par la réutilisation*) pour définir de nouveaux éléments architecturaux ou même de nouvelles applications. Cette dimension concerne la description des éléments architecturaux suivant deux étapes [Barbier *et al.* 2004] :

1. *L'ingénierie pour la réutilisation :* permet de spécifier les éléments qui sont construits pour la réutilisation de chaque niveau conceptuel. Les éléments réutilisables sont identifiés selon leur niveau conceptuel en utilisant un langage formel, semi-formel ou descriptif et ensuite intégrés dans une bibliothèque associée en choisissant une méthode bien définie (hiérarchisée par type, par taille, par contenu, par la fréquence de réutilisation, etc.). L'ingénierie pour la réutilisation se décline en trois phases :

- *Identification*. Dans cette phase, les besoins de l'utilisateur sont identifiés. Par exemple, dans le niveau conceptuel A_3 on peut identifier le besoin de méta-composants ou de méta-connecteurs.
- *Représentation*. Après que les éléments architecturaux soient identifiés, ils doivent être représentés. Des langages formels, semi-formels ou descriptifs sont utilisés pour la description de ces éléments.
- *Organisation*. Les éléments architecturaux sont alors organisés dans le niveau approprié de la bibliothèque en fonction de leur niveau conceptuel.

2. *L'ingénierie par la réutilisation* : permet de spécifier les éléments qui peuvent être construits par la réutilisation où la recherche et la sélection des éléments réutilisables qui sont faites selon les besoins exprimés par les différentes catégories des utilisateurs. La technique de la recherche dépend fortement de l'organisation des éléments dans la bibliothèque aussi bien que du langage choisi durant l'étape précédente. Les éléments sont adaptés si nécessaire, et intégrés dans l'architecture. L'ingénierie par la réutilisation est réalisée selon trois phases :

- *La recherche et la sélection.* Il s'agit de rechercher un élément architectural dans la bibliothèque.
- *L'adaptation.* Il s'agit d'adapter un élément architectural aux besoins de l'utilisateur. L'adaptation peut être réalisée en utilisant des mécanismes tels que : l'instanciation, la composition, la généralisation/spécialisation, etc. Ces mécanismes représentent des intra-liens entre les mêmes types de concepts.
- *L'utilisation.* Finalement l'élément architectural est opérationnel et peut être intégré au niveau conceptuel approprié.

3.6 Conclusion

Nous avons présenté dans ce chapitre le méta-modèle d'architecture baptisé C3, que nous proposons pour prendre en compte la problématique de description des architectures logicielles suivant différentes vues hiérarchiques. Nous avons mis en avant dans C3 la nécessité de disposer d'une nouvelle structure pour le concept de connecteur, considéré comme une entité de première classe au même niveau qu'un composant. Dans C3 nous proposons un ensemble de types de connecteur permettant la description des différentes hiérarchies. Nous avons souligné à travers C3, la nécessité de considérer les différents niveaux de modélisation d'une architecture logicielle, et la nécessité de gérer les mécanismes d'interactions ainsi que les configurations relatives aux différents niveaux d'abstraction, de manière uniforme.

Nous avons introduit le formalisme MY qui décrit les concepts architecturaux selon trois branches : composant, connecteur et configuration. Des liens intra et inter permettent de gérer les différentes dépendances qu'entretiennent les trois aspects de MY. En outre, MY est multi-hiérarchies et multi-abstractions. Cependant, MY est utilisé pour décrire sous forme modulaire et réutilisable des architectures logicielles. Nous avons également mis en avant l'importance de la réutilisation pour garantir un gain important au niveau du coût de développement.

Jusqu'à présent, nous n'avons proposé aucune démarche pour aider les architectes à exploiter C3. Une démarche est pourtant nécessaire pour mettre en avant l'importance de notre méta-modèle dans le contexte de description des architectures logicielles à base de composants. Dans le chapitre suivant, nous allons proposer un profil UML2.0 pour notre méta-modèle C3 en se basant sur des fragments du méta-modèle UML2.0 qui seront identifiés à l'avance. Deux expérimentations sur le modèle C3 seront également introduites.

CHAPITRE 4

Réalisations et Expérimentations

4.1 Introduction

Dans ce chapitre nous définissons un langage intermédiaire sous forme d'un profil UML, afin de faciliter la correspondance entre des modélisations de systèmes en UML et le méta-modèle C3. La définition d'un profil exige au préalable : la description du domaine couvert par le profil et l'identification du sous-ensemble UML2.0 concerné. Le domaine couvert par notre profil, à savoir C3, est décrit sous forme d'un méta-modèle (cf. chapitre 3). Le sous ensemble d'UML2.0 concerné par notre profil sera identifié conformément aux choix et aux stratégies qui seront évoqués dans la section 4.5.

La deuxième section de ce chapitre présente les concepts de bases du méta-modèle UML2.0. La troisième présente et discute les différentes stratégies potentielles permettant de modéliser en UML des architectures logicielles. La quatrième section présente les aspects fondamentaux d'OCL nécessaires à la compréhension du profil proposé. La cinquième section introduit les stéréotypes, les valeurs marquées et les contraintes OCL formant notre profil UML2.0 pour le méta-modèle C3 : C3-UML. Une expérimentation autour de ces propositions sera présentée dans la sixième section. Dans cette expérimentation, nous présentons deux exemples pour illustrer comment le méta-modèle C3 peut être utilisé pour décrire des systèmes logiciels. Pour cela, nous avons choisi deux applications très répandues dans la communauté de l'architecture logicielle : un exemple de type *Client-Serveur* et un exemple de type *Pipe-Filter*. Tout d'abord, nous décrivons chaque application suivant l'approche C3 et ensuite nous l'implémentons avec Java après sa traduction en UML2.0. En dernier lieu, nous comparons notre travail avec l'ADL ArchJava [ArchJava 2004] [Aldrich *et al.* 2002] qui offre les moyens de raffiner une architecture abstraite jusqu'au code Java. Enfin, la dernière section conclut ce chapitre.

4.2 Le méta-modèle UML 2.0

UML (*Unified Modeling Language*) est un langage de modélisation graphique, semi-formel et largement répandu dans le monde industriel. UML est adopté comme un standard

industriel de facto. UML1.x est un langage de modélisation orienté objet [OMG 2003b]. Il propose neuf diagrammes différents, appelés aussi vues. UML1.x est basé sur un méta-modèle unique.

UML2.0 [OMG 2005] propose un modèle de composants permettant de définir les spécifications des composants, ainsi que l'architecture des systèmes à développer. UML2.0 décrit treize types de diagrammes officiels. UML2.0 représente le changement le plus important qu'ait jamais connu UML, les modifications les plus profondes concernent le méta-modèle d'UML.

Cette section comporte trois volets. Le premier volet est consacré à l'étude du modèle de composants préconisé par UML2.0. Quand au deuxième volet, il présente l'organisation générale du méta-modèle UML2.0. Le dernier volet présente les fragments concernés pour adapter UML2.0 au méta-modèle C3.

4.2.1 Modèle de composants UML2.0

4.2.1.1 Composant

Un *composant* UML2.0 représente une partie modulaire d'un système qui encapsule son contenu et qui est remplaçable au sein de son environnement. Un composant définit un comportement en termes d'interfaces fournies ou requises. Ces interfaces contiennent un ensemble d'opérations, d'attributs et de contraintes OCL2.0 (langage formel d'expression de contraintes pour UML) [OMG 2003c]. Un composant UML2.0 est voué à être déployé un certain nombre de fois, dans un environnement non connu à priori lors de la conception. La Figure 4.1 montre une présentation possible d'un composant doté d'une interface offerte et d'une interface requise.

Figure 4.1- Représentation d'un composant doté de deux interfaces.

4.2.1.2 Port

Un composant UML2.0 interagit avec son environnement par l'intermédiaire de points d'interaction appelés ports. Un *port* est une caractéristique structurelle d'un classificateur encapsulé qui spécifie un point d'interaction entre le classificateur et son environnement ou entre un classificateur et ses parties internes appelées *parts*. Seules les classes et les composants peuvent contenir des ports. Les interfaces requises ou offertes peuvent être associées aux ports, afin de spécifier les opérations attendues de l'environnement ou offertes par le classificateur.

Le comportement d'un port est issu de la composition des comportements de ses interfaces. Un tel comportement est décrit à l'aide d'une machine à états de description de protocoles. Le comportement interne du composant ne doit être ni visible, ni accessible autrement que par ses ports. Les ports installés sur des composants ou des classes peuvent être fournis ou requis. La Figure 4.2(a) montre un composant appelé Serveur1 doté d'un ou plusieurs ports (*multiplicité 1..**). Une interface de type Serveur1 peut avoir un ou plusieurs ports P. La Figure 4.2(b) montre un composant Serveur2 dont le port P est doté d'une interface offerte « *request* » et d'une interface requise « *response* ».

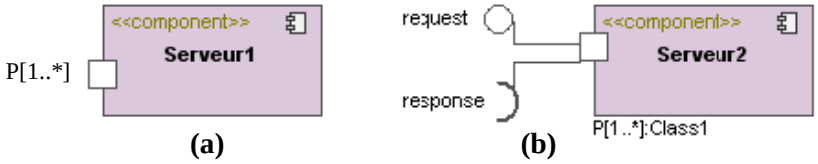


Figure 4.2- Représentation d'un port avec ou sans interfaces.

4.2.1.3 Structure composite

Il existe deux types de modélisation de composants dans UML2.0 : le composant atomique (ou *basique*) et le composant composite (structure *composite*). La première catégorie définit le composant comme étant un élément exécutable du système. La deuxième catégorie étend la première en définissant le composant comme étant un ensemble cohérent des parties appelées *parts*. Chaque partie représente une instance d'un autre composant. La Figure 4.3 montre la vue externe dite boîte blanche d'un composant FusionEtTri permettant de fusionner et trier deux flots d'entrée. Le composant FusionEtTri offre deux ports *Entree1* et *Entree2* et exige un port *Sortie*.

Figure 4.3- Vue externe d'un composant FusionEtTri.

La Figure 4.4 montre la vue interne dite encore boîte blanche d'un composant FusionEtTri. Ce dernier est composé de deux sous-composants : Fusion et Tri. La direction des connecteurs de délégation indique que les deux ports *entree1* et *entree2* sont typés avec des interfaces offertes et le port *sortie* est typé avec une interface requise. Les parties Fusion et Tri sont connectées par un connecteur d'assemblage non nommé.

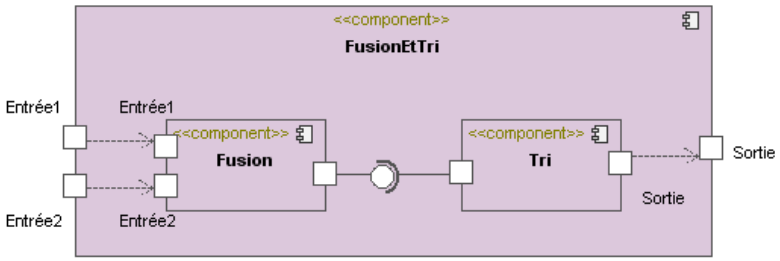


Figure 4.4- Vue interne d'un composant FusionEtTri.

4.2.1.4 Connecteur

En UML2.0 La connexion entre les ports *requis* et les ports *fournis* se fait au moyen de connecteurs. Il existe deux types de connecteurs : le connecteur de *délégation* et le connecteur *d'assemblage*.

i. *Le connecteur de délégation* est un connecteur qui relie le contrat externe d'un composant (spécifie par ses ports) à la réalisation de ce comportement par les parties internes du composant. Il permet de lier un port du composant composite vers un port d'un composant situé à l'intérieur du composant composite, par exemple, relier un port requis à un autre port requis. Un connecteur de délégation doit uniquement être défini entre les interfaces utilisées ou des ports de même type, c'est-à-dire entre deux ports ou interfaces fournies ou entre deux ports ou interfaces requis par le composant.

ii. *Le connecteur d'assemblage* est un connecteur entre deux composants qui définit qu'un composant fournit le service qu'un autre composant requiert. Un connecteur d'assemblage doit uniquement être défini à partir d'une interface requise ou d'un port vers une interface fournie ou un port.

La Figure 4.5 montre un connecteur d'assemblage entre le composant C1 et le composant C2, et un connecteur de délégation entre le connecteur C3 et C4.

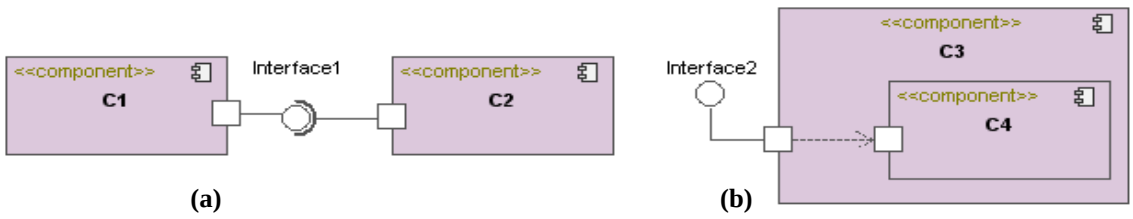


Figure 4.5- (a). Connecteur d'assemblage, (b) Connecteur de délégation.

4.2.2 Organisation générale du méta-modèle UML2.0

UML est composé de deux standards : UML2.0 Superstructure, qui définit la vision utilisateur et UML2.0 Infrastructure, qui spécifie l'architecture de méta-modélisation d'UML

ainsi que son alignement avec MOF (*Meta Object Facility*) [OMG 2002a]. Dans la suite de cette section, nous nous intéressons à UML 2.0 Superstructure notée tout simplement UML2.0. Tout d'abord, nous présentons le canevas de description des parties du méta-modèle UML2.0. Ensuite, nous décrivons l'architecture générale du méta-modèle UML2.0.

4.2.2.1 Canevas de description

Chaque élément du langage UML2.0 (classe, association, attribut, opération, composant, etc.) est représenté dans le méta-modèle UML2.0 par la méta-classe correspondante (*Class*, *Association*, *Attribute*, *Operation*, *Component*, etc.). Les relations entre éléments sont représentées par des méta-associations. Par exemple, une classe UML 2.0 peut comporter plusieurs attributs. Un tel fait est représenté par une méta-association (précisément une méta-composition) entre les deux méta-classes *Attribute* et *Class* dans le méta-modèle UML2.0. Une description complète du méta-modèle UML2.0 peut être trouvée dans [OMG 2005].

4.2.2.2 Architecture générale du méta-modèle UML2.0

Le méta-modèle UML2.0 est extrêmement complexe, avec une cinquantaine de packages et une centaine de méta-classes. Au plus haut niveau, le méta-modèle UML2.0 est découpé en trois parties :

- La partie *Structure* définit les concepts statiques (classe, attribut, opération, instance, composant, package, etc.) nécessaires aux diagrammes statiques, tels que les diagrammes de classes, de composants et de déploiement ;
- La partie *Behavior* définit les concepts dynamiques (interaction, action, état, etc.) nécessaires aux diagrammes dynamiques, tels que les diagrammes d'activités, d'état et de séquences ;
- La partie *Supplement* définit les concepts additionnels, tel que les types primitifs et les templates. Cette partie définit aussi le concept de profil.

UML2.0 introduit une nouvelle relation entre packages : la relation *merge* [Blanc 2005]. Cette relation est fortement utilisée par UML2.0. Elle nécessite un package source et un package cible. Le package cible de la relation est appelé package « mergé » et le package source package « mergeant ».

4.2.3 Fragments d'UML2.0 pertinents

Rappelons que l'objectif recherché de cette partie de la thèse est d'établir un profil UML 2.0 pour le méta-modèle C3. Tout d'abord, nous allons présenter la partie du méta-modèle UML2.0 qui contient les méta-classes relatives aux profils, ensuite, identifier les méta-classes cibles du méta-modèle UML2.0 pour stéréotyper les concepts de C3.

4.2.3.1 Profils UML2.0

UML 2.0 facilite la création de nouveaux profils en simplifiant la définition des concepts de profils et de stéréotypes dans le méta-modèle. La Figure 4.6 illustre la partie du méta-modèle UML 2.0 qui contient les méta-classes relatives aux profils.

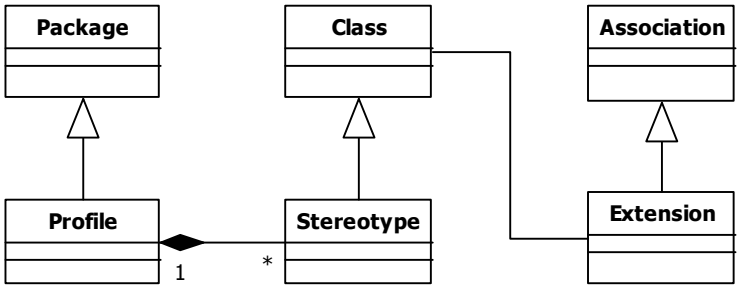


Figure 4.6 – Les profils dans UML 2.0.

Dans ce méta-modèle, les définitions des nouveaux profils sont considérées comme des packages. Ceci explique la relation d'héritage entre les deux méta-classes *Profile* et *Package*. La méta-classe *Stereotype* hérite de la méta-classe *Class*. Ceci a comme conséquence que les définitions des stéréotypes sont considérées comme étant des classes UML2.0.

Les classes étendues par les stéréotypes introduisent de nouvelles méta-classes dans le méta-modèle. Ce qui représente le lien – via la méta-classe *Extension* – entre les deux méta-classes *Stereotype* et *Class*. Sachant que la méta-classe *Class* représente aussi bien le concept de classe que celui de méta-classe.

4.2.3.2 Concepts structuraux de C3 et UML2.0

Dans cette section, nous nous interrogeons sur les concepts les plus proches d'UML2.0 permettant d'être utilisés comme cibles pour stéréotyper les concepts structuraux de C3 : composant, connecteur et configuration.

Les concepts structuraux de C3 à savoir composant, connecteur et configuration sont considérés comme des *types*. De plus, ils sont traités comme des entités occupant le même niveau conceptuel (entité de première classe). Enfin, la vision externe de ces trois concepts est basée sur un ensemble de *ports*. Les caractéristiques relatives aux concepts de composant et connecteur de C3 sont similaires aux concepts composant et classe d'UML2.0. En effet, en UML2.0, composant et classe sont considérés comme des classificateurs (*classifier*). Sachons qu'un classificateur permet de classifier un ensemble d'objets. En outre, en UML2.0, composant et classe sont considérés comme des classificateurs encapsulés sur lesquels on peut installer des ports au sens d'UML2.0.

Bien que les deux concepts composant et classe d'UML2.0 possèdent le même pouvoir expressif, ils sont utilisés comme base pour stéréotyper respectivement les deux concepts composant et connecteur de C3. Ceci permet de distinguer les deux constructions

(composant et connecteur) de C3 de point de vue utilisation. En effet, composant et classe d'UML2.0 possède deux icônes différentes.

4.2.3.2.1 Classe et composant comme classificateurs

La Figure 4.7 illustre comment les deux concepts classe et composant sont considérés comme des classificateurs. En effet, les deux méta-classes *Class* et *Component* héritent d'une façon indirecte de la méta-classe *Classifier*. Celle-ci est une méta-classe abstraite qui représente des classificateurs tels que Class, Component, Interface, DataType, UseCase, etc. De plus, les deux méta-classes Class et Component dérivent indirectement de la méta-classe StructuredClassifier. Ceci précise que Class et Component peuvent contenir des connecteurs. Enfin, les deux méta-classes Class et Component provient de la méta-classe EncapsulatedClassifier. Ceci précise que Class et Component peuvent contenir des ports. Dans la suite, on va détailler respectivement les concepts connecteur, port, classe et composant.

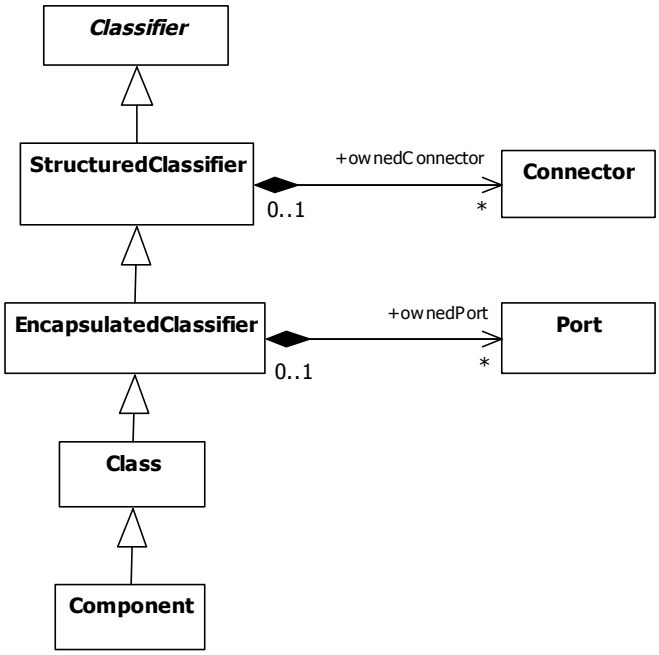


Figure 4.7- Classe et composant comme classificateurs.

4.2.3.2.2 La méta-classe Connector dans le méta-modèle UML2.0

Dans ce méta-modèle (cf. Figure 4.8), la méta-classe Connector hérite de la méta-classe Feature. Cela signifie qu'un connecteur est une caractéristique (*feature*) qui peut être attachée à des instances appartenant à des classificateurs. La méta-classe Connector est reliée à la méta-classe ConnectorEnd par une méta-association qui précise d'un connecteur doit contenir au moins deux instances de types ConnectorEnd.

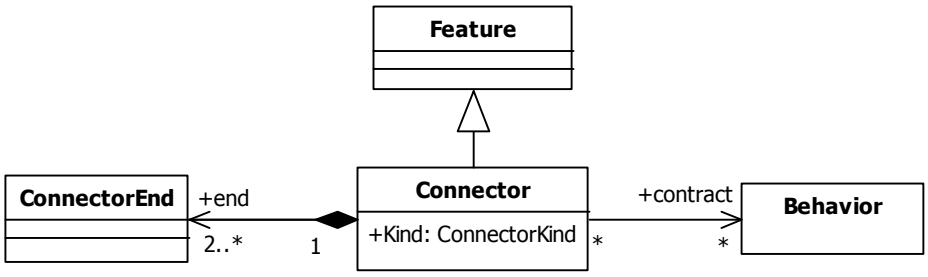


Figure 4.8- Structure de la méta-classe Connector.

4.2.3.2.3 La méta-classe Port dans le méta-modèle UML2.0

La Figure 4.9 illustre la partie du méta-modèle UML2.0 qui définit la méta-classe *Port*. La méta-classe Port hérite à la fois de la méta-classe *StructuralFeature* et de la méta-classe *ConnectableElement*. Un port est considéré comme une caractéristique (*feature*) structurelle, et il peut être connecté par des connecteurs. De plus, la méta-classe Port est reliée à la méta-classe Interface par deux méta-associations, qui précisent qu'un port peut identifier des interfaces requises ou offertes. Enfin, la méta-classe Port est reliée à la méta-classe ProtocolStateMachine par une méta-association, qui précise que le comportement d'un port peut être décrit par une machine à état de description de protocoles.

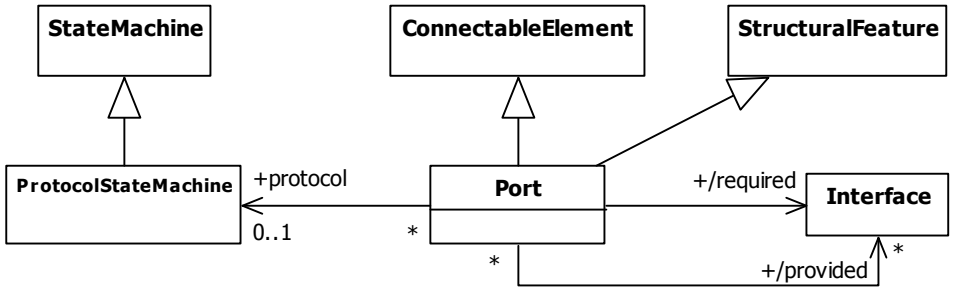


Figure 4.9- La méta-classe Port.

4.2.3.2.4 La méta-classe Class dans le méta-modèle UML2.0

La méta-classe *Class* possède des caractéristiques structurelles et comportementales. Ceci est traduit par deux méta-associations entre la méta-classe Class et respectivement la méta-classe *Property* et la méta-classe *Operation*. La Figure 4.10 illustre la partie du méta-modèle UML2.0 qui définit la méta-classe Class.

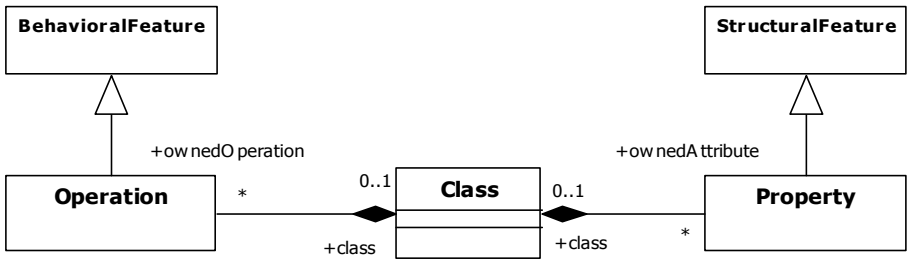


Figure 4.10- Structure de la méta-classe Class.

4.2.3.2.5 La méta-classe Component dans le méta-modèle UML2.0

La Figure 4.11 illustre la partie du méta-modèle UML2.0 qui définit la méta-classe *Component*. La métaclasse Component est reliée à la méta-classe Interface par deux méta-associations. Ainsi, un composant a des interfaces offerts et/ou requises. La méta-classe Component est reliée à la méta-classe Realization par une méta-association qui précise qu'un composant peut être réalisé par une ou plusieurs instances de type Realization. La méta-classe Realization est reliée à la méta-classe Classifier par une méta-association qui précise qu'une Realization est décrite par un classifier.

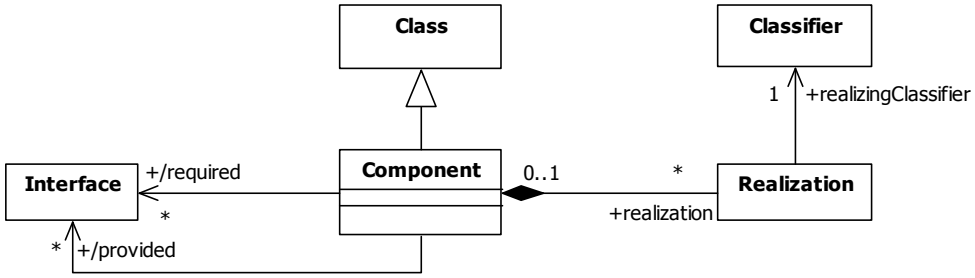


Figure 4.11- Structure de la méta-classe Component.

4.3 Architectures logicielles modélisées en UML

L'architecture de méta-modélisation en quatre niveaux appliquée au langage UML introduit trois stratégies potentielles pour modéliser des architectures logicielles en UML [Robbins *et al.* 1998] [Medvidovic *et al.* 2002]. Ces stratégies sont :

- l'utilisation UML2.0 en tant que tel ;
- l'utilisation des mécanismes d'extensibilité d'UML2.0 pour contraindre le méta-modèle UML afin de l'adapter aux concepts architecturaux ;
- l'augmentation du méta-modèle UML afin de supporter directement les concepts architecturaux.

Chaque stratégie a des avantages et des inconvénients. Dans la suite, nous allons présenter et évaluer ces trois stratégies en s'inspirant des travaux effectués par [Medvidovic *et al.* 2002] [Zarras *et al.* 2001] [Roh *et al.* 2004] [Goulão et Abreu 2003] [Kandé et Strohmeier 2000] et [Ivers *et al.* 2004].

4.3.1 Utilisation directe d'UML

La question qu'UML est un ADL, et est-il approprié pour cette fin, a fait l'objet de plusieurs études et débats [Booch *et al.* 1999]. Sous la définition la plus large de l'architecture logicielle donnée au chapitre 1, il est clair que même les versions antérieures d'UML peuvent être utilisées comme une notation de modélisation d'architectures [Hilliard 1999] [Hofmeister *et al.* 1999]. UML a émergé presque en même temps que la première génération d'ADLs et malgré ses lacunes constatées lors de la modélisation de préoccupations architecturales critiques [Medvidovic et Taylor 2000] [Medvidovic *et al.* 2002], il a été rapidement et largement adopté. UML a continué à évoluer. Depuis 1997 seulement, il a subi quatre modifications majeures: version UML 1.0, 1.1, 1.3 et 2.0. En outre, la sortie de la version UML2.0, avec au moins quelques enrichissements destinés à améliorer la modélisation d'architecture logicielle [Booch *et al.* 2005].

Cette stratégie (utilisation directe d'UML « As Is ») consiste à utiliser les constructions offertes par UML pour représenter les concepts architecturaux venant des ADLs tels que composant, connecteur, port, rôle et configuration. Garlan et al. [Garlan *et al.* 2000] [Garlan *et al.* 2002] proposent et évaluent quatre approches permettant de représenter les principaux concepts architecturaux en UML1.*x*. Ces approches sont centrées autours de la représentation des *types* de composant et des *instances* de composant en UML 1.*x*. Ainsi, ces auteurs proposent les quatre options suivantes :

- *classes et objets* : Les types de composants sont représentés par des classes UML1.x. et les instances de composants par des objets,
- *classes et classes* : les types de composants et les instances de composants sont représentés par des classes UML1.x,
- *composants UML 1.x* : les types de composants sont représentés par des composants UML 1.x et les instances de composants par des instances de composants UML1.x,
- *subsystems* : les types de composants sont représentés par des subsystems UML et les instances de composants par des instances de subsystems.

En fonction de l'option, les autres concepts architecturaux comme port, connecteur, rôle et configuration peuvent être représentés par d'autres constructions UML adéquates à l'option choisie. Par exemple, le concept port dans la première option peut être représenté par plusieurs choix comme interface, attribut, classe contenue (*contained class*).

L'utilisation d'UML en tant que tel est également employée pour représenter des ADLs particuliers. Par exemple, dans [Medvidovic et Taylor 2000], UML1.*x* est utilisé pour modéliser l'ADL C2. Dans [Ivers *et al.* 2004], UML 1.*x* est utilisé pour modéliser des concepts architecturaux issus de l'ADL ACME tels que composant, port, connecteur, rôle et attachement. Les auteurs de cette étude utilisent avec profit les nouveaux concepts proposés

par UML2.0 tels que composant, port et connecteur d'assemblage. L'avantage majeur de l'utilisation d'UML en tant que tel pour la modélisation des architectures logicielles décrites par des ADLs est la compréhension de cette modélisation par n'importe quel utilisateur d'UML. De plus, une telle modélisation peut être manipulée par des ateliers UML supportant les autres étapes du développement : conception et implémentation. Quant à l'inconvénient inhérent à cette stratégie, il concerne l'incapacité d'UML en tant que tel – notamment UML1.x – à traduire d'une façon *explicite* les concepts architecturaux.

4.3.2 Utilisation des mécanismes d'extensibilité

UML est un langage de modélisation « *généraliste* » pouvant être adapté à chaque domaine grâce aux mécanismes d'extensibilité offerts par ce langage tels que stéréotypes, valeurs marquées (*tagged value*) et contraintes. Les extensions UML ciblant un domaine particulier forment des profils UML. Les mécanismes d'extensibilité offerts par UML permettent d'étendre UML sans toucher au méta-modèle UML (*lightweight extension).*

Plusieurs travaux permettent d'adapter aussi bien UML1.x qu'UML2.0 aux architectures logicielles. Ces travaux peuvent être classés en deux catégories : des travaux visant des ADLs particuliers et d'autres ciblant des ADLs génériques c'est-à-dire comportant des concepts architecturaux communs à tous les ADLs.

Medvidovic et Taylor [Medvidovic et Taylor 2000] proposent trois profils UML1.x – précisément UML1.3 – pour les ADLs C2, Wright et Rapide. Nous nous sommes inspirés de cette recherche intéressante pour l'élaboration de notre profil UML2.0 pour le méta-modèle C3. Les auteurs de [Goulão et Abreu 2003] établissent un profil UML2.0 pour l'ADL ACME [Garlan *et al.* 2000]. Celui-ci est considéré comme un ADL pivot permettant l'interopérabilité entre les différents ADLs. Ainsi, il regroupe des concepts communs et minimaux à tous les ADLs. Les auteurs de [Roh *et al.* 2004] signalent quelques faiblesses de ce travail liées notamment à la représentation proposée du connecteur (au sens d'UML2.0) et proposent un ADL générique sous forme d'un profil UML2.0. Dans ce travail, nous notons l'utilisation des collaborations UML2.0 pour représenter des connecteurs ADLs. De plus, un autre aspect intéressant se dégage : type et instance de connecteurs sont modélisés par deux stéréotypes. En effet, le type de connecteur est défini comme stéréotype à base de méta-classe *Collaboration* d'UML2.0. Et l'instance de connecteur est définie comme stéréotype à base de méta-classe *Connector* d'UML2.0. Mais ce travail ne traite pas les aspects comportementaux des ADLs.

L'utilisation des profils pour adapter UML au domaine des architectures logicielles a comme avantage important d'*expliciter* la représentation des concepts architecturaux sans toucher au méta-modèle UML. De plus, ces profils peuvent être manipulés par des ateliers UML supportant le concept profil. Actuellement, ces ateliers sont de plus en plus nombreux tel que Softeam UML Modeler (http://www.objecteering.com) et IBM Rational Software Modeler (http://www.rational.com). La contrepartie vis-à-vis des utilisateurs est de connaître notamment les contraintes décrites souvent en OCL – associées aux stéréotypes formant ces profils.

4.3.3 Augmentation du méta-modèle UML

Cette stratégie consiste à ajouter de nouveaux éléments de modélisation (de nouvelles constructions syntaxiques) en modifiant directement le méta-modèle UML (*heavyweight extensibility mechanism*). Ceci donne un nouveau langage de modélisation supportant d'une façon native les concepts architecturaux.

Les auteurs de [Enrique et Martinez 2003] augmentent le méta-modèle UML afin de supporter directement les concepts architecturaux. L'augmentation du méta-modèle d'UML pour couvrir des besoins architecturaux, se heurte aux deux inconvénients majeurs suivants :

- la nouvelle version d'UML devient de plus en plus complexe. Ceci a un impact sur la facilité d'utilisation de ce langage.
- la nouvelle version d'UML perd son caractère standard et par conséquent elle devient incompatible avec les ateliers existants.

4.4 OCL

Dans cette section, nous présentons les aspects fondamentaux d'OCL nécessaires à la compréhension du profil UML2.0 pour le méta-modèle C3 proposé. Depuis sa version 1.3, UML intègre un langage de spécification de contraintes OCL (*Object Constraint Langage*) [Warmer et Kleppe 2003]. Ce langage permet l'écriture d'expressions booléennes sans effet de bord qui restreignent le domaine de valeurs d'un modèle UML. Ces expressions peuvent être attachées à n'importe quel élément UML. De telles expressions OCL peuvent aussi s'appliquer au niveau du méta-modèle [OMG 2005] et permettent de décrire les règles de bonne utilisation (*Well Formedness Rules : WFR*). De plus, les expressions OCL sont utilisées pour définir des stéréotypes. En outre, les expressions OCL favorisent la conception par contrat (*Design by Contract*) préconisée par l'auteur d'Eiffel [Meyer 1992] [Meyer 2000].

La nouvelle spécification d'OCL notée OCL2.0 [OMG 2003c] fait partie intégrante d'UML2.0. Le langage OCL2.0 – noté dans cette thèse OCL – ajoute aux anciens types de contraintes (*inv, pre, post*) des nouveaux types de contraintes (*init, def, body*). Contrairement aux anciens types de contraintes, les nouveaux sont considérés comme des expressions non forcément booléennes. Les contraintes d'utilisation qui accompagnent les stéréotypes et qui forment notre profil C3-UML sont toutes des contraintes booléennes de type *inv* (*invariant*). Sachant qu'une contrainte de type *inv*, attachée à une méta-classe, doit être observée pour toutes les instances de cette méta-classe.

4.4.1 Eléments du Langage OCL

4.4.1.1 Contrainte

Les contraintes OCL sont utilisées en UML pour spécifier des restrictions sur les éléments des différents modèles. Elles peuvent être attachées à tous type d'éléments du modèle et doivent être respectées dans toute modification ou raffinement d'un diagramme.

4.4.1.2 Contexte

Toute contrainte OCL est liée à un contexte spécifique : l'élément auquel la contrainte est attachée qu'on peut spécifier explicitement à l'intérieur d'une expression à l'aide du mot clé « *self* ».

- Si le contexte est un type, alors self se rapporte à un objet de ce type.
- Si le contexte est une opération, alors self désigne le type qui possède cette opération.
  - **Exemple :** soit la classe compte :

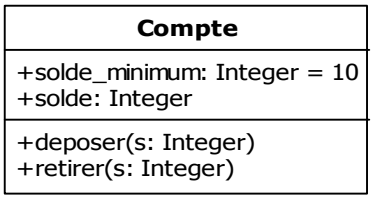


Figure 4.12 – Description de la classe compte.

Les contraintes attachées à la classe Compte illustrée par la Figure 4.12 peuvent être :

```
context Compte
        Inv solde_minimum : self.solde ≥ Compte.solde_minimum
context Compte :: deposer(s : Integer)
        pre : s > 0
        post : self.solde = self.solde@pre+s
```

L'opérateur **@pre** dénote l'état précédent c'est-à-dire avant l'exécution de l'opération.

4.5 Réalisation de C3 en UML 2.0

Les nouveaux concepts et les modifications apportées à d'autres dans UML2.0 fournissent un vocabulaire riche pour documenter une architecture logicielle et permettre de résoudre la plupart des problèmes dus à l'utilisation des précédentes versions du langage UML. Avec la version UML2.0, il y a plusieurs manières de représenter un concept architectural. De ce fait, plusieurs stratégies de transformation peuvent être utilisées pour représenter les différents concepts de C3.

4.5.1 Stratégies de projection

La projection d'un ADL vers UML peut être réalisée soit manuellement en examinant toutes les notations d'UML, et ensuite choisir les correspondances appropriées soit semi-automatiquement en définissant une correspondance entre leurs méta-modèles respectifs.

Dans la stratégie semi-automatique, on fait correspondre les notions d'un méta-ADL au MOF et on obtient par instanciation la correspondance de leurs instances respectives. La Figure 4.13 montre les deux stratégies de projections.

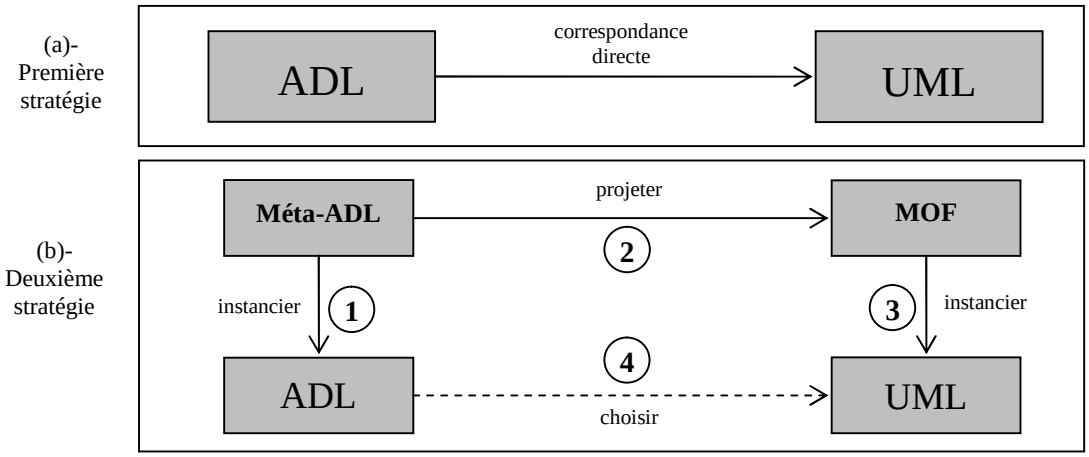


Figure 4.13- Stratégies de projection d'un ADL vers UML.

La première stratégie (cf. Figure 4.13-a) est basée sur l'examen de toutes les notations et sémantiques d'UML ensuite de faire correspondre des notations et sémantiques inhérentes à un ADL donné. Si des notations appropriées ne sont pas trouvées, des mécanismes d'extensions d'UML (stéréotypes, valeurs marquées, contraintes) sont utilisés. Cependant, cette stratégie a un certain nombre d'inconvénients tels que :

- La projection est faite manuellement. Une méthode bien établie qui guide le processus n'existe pas actuellement, mais certains travaux ont proposé un nombre de critères pour mieux justifier leurs choix [Garlan *et al.* 2002].
- Cette stratégie exige que tout le méta-modèle d'UML soit examiné pour chaque notation de l'ADL.
- Un long processus de recherche est exigé chaque fois qu'un nouvel ADL est considéré pour être projeté vers UML.

La deuxième stratégie (cf. Figure 4.13-b) consiste à établir une correspondance entre les concepts architecturaux et objets au niveau méta (MOF) afin de réduire le nombre de concepts à traduire. Pour parvenir à cet objectif, nous devons d'abord créer un Méta-ADL pour l'architecture logicielle et ensuite projeter les éléments du méta-ADL vers le MOF. Cette stratégie de projection est réalisée en quatre étapes :

1- *instancier* le Méta-ADL pour obtenir l'ADL souhaité (*i.e.*, celui que nous voulons projeter vers UML),

2- *projeter* chaque élément du Méta-ADL vers un élément de MOF,

3- *instancier* chaque élément de MOF pour obtenir un ou plusieurs éléments UML, ceci permis d'obtenir plus d'un choix pour chaque élément de l'ADL,

4- *choisir* le concept UML le plus adéquat pour notre ADL.

Parmi les avantages de cette stratégie nous avons :

- La correspondance des concepts traduits au niveau méta réduit considérablement leur nombre.
- Le passage d'un ADL vers le Méta-ADL est naturel, inhérent et facile, dans la mesure où les deux utilisent des éléments architecturaux cohérents.
- Le passage de MOF à UML existe déjà, et il est bien défini par l'OMG.
- La sélection du concept cible d'UML est plus facile vu que le nombre de choix est limité.

Dans ce qui suit, nous allons décrire le processus de projection de C3 vers UML suivant la deuxième stratégie. Cependant, cette stratégie exige la définition d'un niveau plus élevé d'abstraction pour l'architecture logicielle. Pour ce faire nous utilisons le modèle MADL. Plus de détail en Annexe B.

4.5.2 Processus de projection de C3 vers UML

4.5.2.1 Instanciation de MADL par C3

La première étape du processus (Figure 4.13-b) est l'instanciation de MADL par C3, Chaque élément de C3 doit être conforme à un élément de MADL. La Figure B.1 de l'Annexe B montre comment MADL a été instancié par C3. Les composants et les configurations, les connecteurs et les connexions, les ports, et les styles sont obtenus grâce aux instanciations successivement de *Méta-Composant*, *Méta-Connecteur*, *Méta-Interface* et *Méta-Architecture*. Le Tableau 4.1 résume la relation d'instanciation de MADL par C3.

Concepts C3	Concepts MADL
Elément-Architectural	Méta-Composant
Composant	Méta-Composant
Connecteur	Méta-Connecteur
Configuration	Méta-Composant
Interface	Méta-Interface

Port	Méta-Interface
Glu	Méta-Connecteur
Service (service de composant)	Méta-Interface
ServiceC (service de connecteur)	Méta-Interface
Connexion	Méta-Connecteur
Use	Méta-Connecteur
Style Architectural	Méta-Architecture

Tableau 4.1- L'instanciation de MADL par C3.

4.5.2.2 Projection des concepts de MADL vers MOF

En principe, MOF peut être utilisé pour définir un méta-modèle pour l'architecture logicielle. Cependant, MOF soufre d'un certain nombre de limites concernant la description architecturale. En effet, MOF contient des concepts de base pour définir des méta-entités (*e.g., Class* du MOF) des méta-relations (*e.g., Association* du MOF) et des paquetages jouant le rôle de conteneurs pour ces méta-entités et méta-relations. On peut noter qu'aucune relation dans le MOF ne permet d'indiquer qu'une architecture est définie par une méta-architecture.

Pour cette raison, nous somme orienté vers l'utilisation de MADL, qui représente l'équivalent du MOF pour l'architecture logicielle. Notons que chaque élément de MADL est représenté comme une sous-classe d'un élément de MOF qui a une sémantique similaire (cf. Figure B.3 de l'Annexe B). La raison de ce choix de représentation et de ne pas violer la sémantique du MOF en changeant sa structure avec de nouvelles notations ou associations. Cependant, si aucune correspondance sémantique n'est trouvée, un nouvel élément est alors présenté en stéréotypant une sous-classe. Le Tableau 4.2 résume la projection de chaque élément de MADL vers sa correspondance dans le MOF.

Concepts MADL	Concepts MOF
Composant	ModelElement
Méta-Composant	Class
Méta-Connecteur	Association
Méta-Interface	Feature
Architecture	Namespace
Méta-Architecture	Package
Méta-Méta-Architecture	Package Stéréotypé

Tableau 4.2- Projection de MADL vers MOF.

4.5.2.3 Instanciation du MOF par UML2.0

L'instanciation du MOF par UML2.0 est faite automatiquement en utilisant les spécifications de l'OMG [OMG 2005] qui décrivent l'instanciation du MOF par UML. Chaque élément d'UML est une instance d'un élément de MOF y compris UML lui-même. Le Tableau 4.3 montre les instances possibles d'UML2.0 pour chaque élément du MOF.

MOF	UML2.0
Class	Class, Component, Stereotype
Association	Class, Association, AssociationClass, CommunicationPath
Feature	Interface, Port, Operation, Property, Connector UML
Package	Package

Tableau 4.3- L'instanciation du MOF par UML2.0.

4.5.3 Critères de choix des règles de passage

Les conventions et règles de passage vers UML2.0, ont été choisies en fonction des critères utilisés par Garlan [Garlan *et al.* 2002] [Ivers *et al.* 2004] et Medvidovic [Medvidovic *et al.* 2002] pour comparer leurs différentes stratégies de représentation des concepts architecturaux en UML. Les deux approches comportent chacune des avantages et des inconvénients résumés ci-dessous en fonction des quatre critères suivants :

Utilisation des nouveaux concepts architecturaux d'UML2.0

La stratégie de Medvidovic d'utiliser UML « tel qu'il est » est pertinente, mais il n'utilise pas les nouveaux concepts d'UML2.0. Garlan dans [Ivers *et al.* 2004] propose de s'appuyer sur cette stratégie pour définir les règles de passage avec la prise en compte des concepts de composants et de connecteurs UML nouvellement introduits dans la version UML2.0.

Séparation visuelle entre composant et connecteur

La volonté de Garlan [Garlan *et al.* 2002] de bien séparer le concept de composant de celui de connecteur en architecture, est une des conséquences du critère de « *lisibilité visuelle* » qu'on exige au niveau architecture. A l'inverse de Medvidovic, nous préférons donc utiliser des concepts différents d'UML pour représenter les composants et les connecteurs. Toutefois, il faut souligner que ceci implique une certaine perte du critère standard d'UML, car ces deux concepts sont encore peu, ou mal définis, par les outils de modélisation d'UML.

Recherche de la même sémantique

La comparaison entre concepts UML et ADLs, découle du critère de Garlan, nommé « *même sémantique* » [Garlan *et al.* 2002]. L'application de cette stratégie implique, par exemple, d'utiliser le concept de composant UML, plutôt que celui de classe UML. Le troisième critère de Garlan permet de vérifier en parallèle si toute la structure et le comportement d'un concept architectural peuvent être représentés grâce à celui d'UML cible choisi. En effet, UML a, par exemple, sa propre vision de ce qu'est un connecteur : juste un lien résultant de l'assemblage de relations de dépendances et d'une interface UML.

Limitation au domaine des composants logiciels

Les règles de passage s'orientent vers le diagramme UML de composants, et permettent ainsi d'obtenir une meilleure compatibilité avec les outils supports. Mais, l'inconvénient principal est que l'on perd la fonctionnalité de génération de code, proposée par de nombreux outils pour les diagrammes de classes. Il faut toutefois modéliser cet inconvénient, car cette génération de code est, la plupart du temps, basique et peu appropriée aux besoins des développeurs.

Le choix est porté sur le concept de « *component* » d'UML2.0 pour la représentation flexible des composants et des configurations C3, mais ces derniers restent bien distincts grâce aux stéréotypes qu'on leur associe. Les connecteurs C3 sont des types explicites et peuvent être réutilisés, ils sont représentés par des classes stéréotypées d'UML2.0. Une telle projection des concepts (composant, connecteur et configuration) est alors très lisible puisque chaque concept est représenté par un concept UML2.0 distinct. Aussi, elle permet de capturer tous les détails des concepts de C3 et de représenter fidèlement la sémantique associée à chaque concept du modèle C3.

4.5.4 Les règles de passage des concepts de C3 vers UML2.0

L'application de notre stratégie de projection nous permet d'obtenir un ou plusieurs concepts UML2.0 correspondant à chaque concept de C3. En se basant sur les critères de choix donnés dans la section 4.5.3, nous obtenons un seul concept d'UML pour chaque concept de C3. Le Tableau 4.4 montre la correspondance de chaque concept de C3 avec un concept d'UML.

Les composants et les connecteurs C3 sont des entités architecturales différentes, le concept de composant tel qu'il est défini dans le standard UML2.0 superstructure convient aussi bien aux composants qu'aux configurations C3 et le concept de classe convient aux connecteurs C3. Dans la suite, nous allons expliciter le passage des concepts C3 vers UML2.0. La section suivante est consacrée à la définition technique du profil C3-UML. Un tel profil comporte un ensemble de stéréotypes appliquées sur des méta-classes UML2.0 et définis par un ensemble de contraintes OCL. Les méta-classes cibles des stéréotypes proposés sont identifiées et justifiées dans les sections précédentes de ce chapitre.

Dans la suite, nous allons établir des stéréotypes permettant de modéliser les concepts structuraux du méta-modèle C3 tels que : composant, connecteur, configuration, port et connexion. Nous allons accorder un soin particulier à l'élaboration des contraintes formelles en OCL associes aux stéréotypes. Ceci permet de mieux cerner le cadre de ces stéréotypes.

Concepts C3	Concepts UML2.0
Elément-Architectural	<<C3Element>> Classe
Propriété	<<C3Propperty>> Attribut
Contrainte	<<C3Constraint>> Contrainte
Implémentation	<<C3Implementation>> Classe
Composant	<<C3Component>> Composant
Connecteur de connexion	<<CC>> Classe
Connecteur de décomposition composition	<<CDC>> Classe
Connecteur de d'expansion compression	<<ECC>> Classe
Connecteur de lien d'identité	<<BIC>> Classe
Connecteur de spécialisation généralisation	<<SGC>> Classe
Connecteur d'instanciation	<<IOC>> Classe
Configuration	<<C3Configuration>> Composant
Interface	<<C3Interface>> Port
Port	<<C3Port>> Interface
Glu	Classe d'association
Service	<<C3Service>> Opération
Connexion	<<C3Connexion>> Connecteur d'assemblage
Use	<<C3Use>> Association

Tableau 4.4- Correspondance entre les concepts de C3 et d'UML2.0.

4.5.4.1 Elément architectural de C3

Le concept de *Class* d'UML correspond au concept élément architectural de C3. Une classe dans UML2.0 est un sous-type de *StructuredClassifier*. Elle peut contenir des propriétés (via l'association *ownedAttributes*) reliées à la classe *Constraint*, qui permet d'attacher une contrainte à n'importe quel élément du modèle et reliée à la classe *Class* par une association *ImplementingClass* qui précise qu'une implémentation est décrite par une classe. Dans le méta-modèle C3 un élément architectural possède une interface, des propriétés, des contraintes et des implémentations.

Stereotype <<C3Element>> :: Class

Context UML :: InfrastructureLibrary :: Core :: Constructs :: Class
inv : self.isStereotyped ("C3Element") implies
self.contents() → oclIsKindOf(C3Interface) → Size()<=1 and
self.ownedAttributs → forAll (a | a. stereotype = C3Property and
self.constraint → forAll (c | c.stereotype = C3Constraint) and
self.implementingClass.stereotype = C3Implementation

Notons qu'une propriété est représenté par un attribut, ayant un type et auquel on peut donner une valeur pour l'instance de l'élément architectural considéré. L'attribut est une propriété structurelle, nous appliquons le stéréotype <<C3Property>> sur la classe *Attribute*.

Aussi, chaque élément architectural peut avoir des contraintes, qui sont des restrictions et conditions qui doivent être vérifiées à tout moment. Nous considérons que le concept de *Constraint* d'UML 2.0 correspond parfaitement au concept de contrainte de C3. Nous utilisons le stéréotype <<C3Constraint>> pour représenter une contrainte architecturale.

Le concept de classe correspond au concept d'implémentation C3 dans UML2.0. Une classe UML2.0 a un nom et peut contenir des attributs. Le concept Implémentation est défini seulement par un <<*C3Element*>> et ne possède pas de ports.

Stereotype <<C3Implementation>> :: Class

Context UML :: InfrastructureLibrary :: Core :: Constructs :: Class
inv : self.isStereotyped ("C3Implementation") implies
self.owner.stereotype = C3Element and
self.ownedPort.IsEmpty ()

4.5.4.2 Interfaces de C3

La classe *Port* d'UML2.0 correspond au concept d'interface de C3 (cf. Figure 4.14). Un port UML est le point de connexion d'un *classifier* vers son environnement et possède des interfaces fournies et requises. C3Interface possède un ensemble non vide de ports.

Stereotype << C3Interface>> :: Port

Target = enum {*composant, connecteur, configuration*}

Context UML :: InfrastructureLibrary :: Core :: Constructs :: Port
inv : self.isStereotyped ("C3Interface") implies
self.owner.steretype = C3Component or
self.owner.stereotype = C3Configuration or
self.owner.stereotype = C3Connector and
self.interface → notEmpty () and
self.interface → forAll (p |p.IsC3Port())

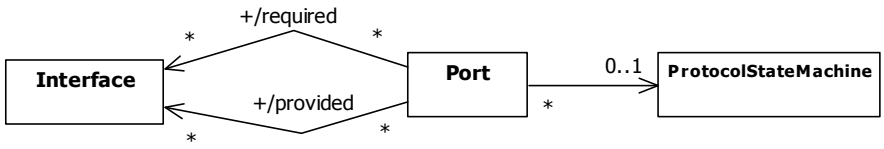


Figure 4.14- La méta-classe *Port* dans le méta-modèle UML2.0.

4.5.4.3 Ports de C3

La classe *interface* d'UML2.0 correspond au Port C3 (cf. Figure 4.15). Dans UML2.0 un port possède une interface fournie et une interface requise, et un seul <<C3ProtocolStateMachine>> est associé au <<C3Port>> traduisant le protocole de fonctionnement attaché à un Port C3.

Stereotype << C3Port>> :: Interface

Mode-Connexion = enum {*synchrone, asynchrone, continu*}

Un port de C3 est défini seulement par une interface

Context UML :: InfrastructureLibrary :: Core :: Constructs :: Interface
inv : self.isStereotyped ("C3Port") implies
self.owner.IsInterface ()
self.ownedOperation → forAll(o|o.formalParameter → isEmpty())
self.ownedAttribute → isEmpty()
self.protocol → size()=1 and
self.protocol → forAll(psm | psm.stereotype= C3ProtocolStateMachine

Un port fourni de C3 est un port C3 qui possède seulement des interfaces fournies.

Context UML :: InfrastructureLibrary :: Core :: Constructs :: Interface
inv : self.isStereotyped ("Port-Fourni") implies
self.IsC3Port () and
self.requis → IsEmpty () and
self.fourni → Size() >=1

Un port requis de C3 est un port C3 qui possède seulement des interfaces requis.

Context UML :: InfrastructureLibrary :: Core :: Constructs :: Interface
inv : self.isStereotyped ("Port-Requis") implies
self.IsC3Port () and
self.fourni → IsEmpty () and
self.requis → Size() >=1

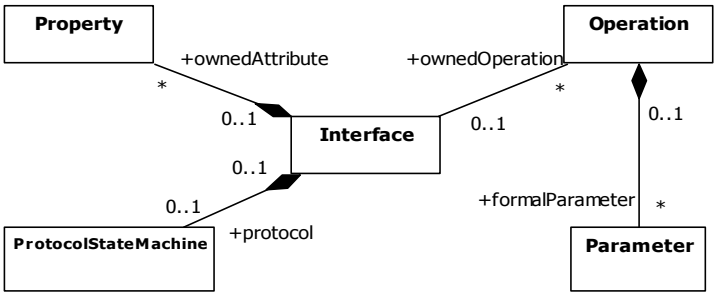


Figure 4.15- La méta-classe *Interface* dans le méta-modèle UML2.0.

4.5.4.4 Composants de C3

Un composant C3 est décrit par un composant UML2.0 (cf. Figure 4.16) stéréotypé par <<C3Component>>. Le stéréotype <<C3Component>> est défini par les contraintes OCL suivantes : Aucune interface offerte ou requise n'est associée à <<C3Component>>, tous les ports associés à <<C3Component>> sont des <<C3Ports>>. Un <<C3Component>> ne possède aucune réalisation, et un <<C3ProtocolStateMachine>> est associé au <<C3Component>> traduisant le protocole de calcul attaché à un composant C3. ComposantC3 qui hérite de *C3Element* possède des *InterfaceComposants*.

Stereotype << C3Component>> :: Component

Context UML :: InfrastructureLibrary :: Core :: Constructs :: Component
inv : self.isStereotyped ("C3Component") implies
self.provided → isEmpty() and self.required → isEmpty()
self.realization → isEmpty()
self.oclIsKindOf(C3Element) and
self.ownedPort → forAll (p | p.IsC3Interface()) and
self.ownedPort.Size () >= 1 and
self.hasNoOtherInterfaces ()
self.stateMachine → size()=1 and
self.stateMachine.oclAsType(ProtocolStateMachine).stereotype=C3ProtocolStateMachine

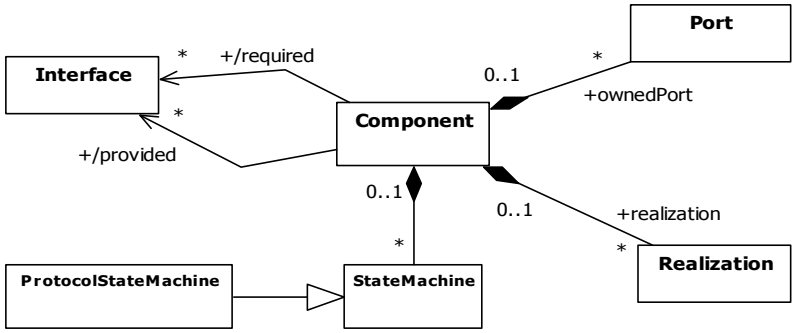


Figure 4.16- La méta-classe *Component* dans le méta-modèle UML2.0.

4.5.4.5 Connecteurs de C3

Un connecteur C3 est décrit par une classe UML2.0 (cf. Figure 4.17) stéréotypée par <<C3Connector>>. Une classe possède zéro ou plusieurs attributs et peut contenir zéro ou plusieurs opérations. La classe représente le type de connecteur et l'instance de classe représente l'instance de connecteur. Ainsi, le concept de classe garantit un connecteur explicite. La définition du composant C3 comme un composant UML et un connecteur C3 comme une classe UML permet de lever l'ambiguïté sémantique et visuelle entre les composants et les connecteurs de C3.

Le stéréotype <<C3Connector>> est défini par les contraintes OCL suivantes : Aucune interface offerte ou requise n'est associée à <<C3Connector>>, tous les ports associés à <<C3Connector>> sont des C3Port, et un <<C3ProtocolStateMachine>> est associé au <<C3Connector>> traduisant le protocole d'interaction attaché à un connecteur C3. Un connecteur C3 qui hérite de *C3Element* définit de zéro à une Glu et peut avoir au minimum deux *C3Interface*.

Dans la spécification OCL des connecteurs C3, nous utilisons <<C3Connecteur>> pour désigner les différents stéréotypes correspondant aux types de connecteurs identifiés dans le chapitre 3 (CC*s*, CDC*s*, ECC, CC*f*, CDC*f*, BIC, SGC et IOC).

Stereotype << C3Connector>> :: Class

Context UML :: InfrastructureLibrary :: Core :: Constructs :: Class
inv : self.isStereotyped ("C3Connector") implies
self.oclIsKindOf(C3Element) and
self.contents() → forAll (g | g.stereotype = C3Glu) and
self.contents() → size() <= 1 and
self.ownedPort → forAll (p | p.IsC3Interface()) and
self.ownedPort.Size () >= 2
self.ownedAttribute → isEmpty() and self.ownedOperation → isEmpty()
self.provided → isEmpty() and self.required → isEmpty()
self.ownedPort → ForAll (p | p.stereotype = C3Port)
self.stateMachine → size()=1 and
self.stateMachine.oclAsType(ProtocolStateMachine).stereotype=C3ProtocolStateMachine

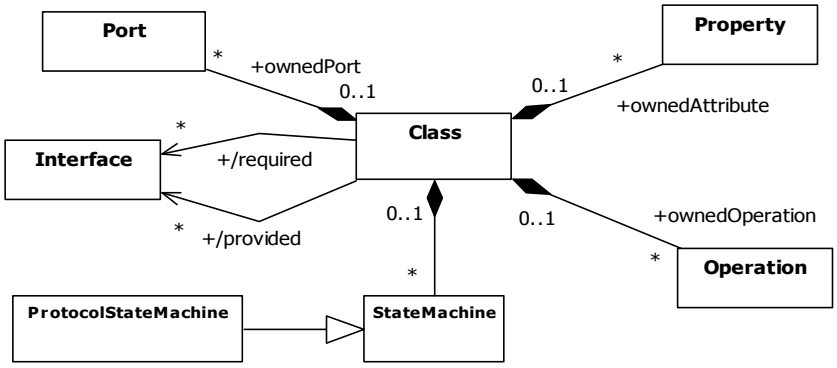


Figure 4.17- La méta-classe *Class* dans le méta-modèle UML2.0.

4.5.4.6 Connexions de C3

Une connexion C3 est décrite par un connecteur d'assemblage UML2.0 (cf. Figure 4.18) stéréotypé par <<C3Connexion>>. Il représente la connexion physique entre les deux interfaces. Le stéréotype <<C3Connexion>> est défini par les contraintes OCL suivantes : Une connexion dans C3 permet de relier deux éléments, un <<C3Connexion>> est un connecteur d'assemblage, les éléments connectables de <<C3Connexion>> sont tous des <<C3Port>>. Le port du connecteur C3 de type fourni est relie avec le port C3 de type requis et vice versa.

Stereotype <<Connexion>> :: Connector

Context UML :: InfrastructureLibrary :: Core :: Constructs :: Connector
inv : self.isStereotyped ("C3Connexion") implies
self.kind = # assembly
self.end → size() = 2
self.end → forAll(ce | ce.port → forAll (elt | elt.oclIsKindOf(port) implies
elt.stereotype = C3Port))
self.end → (exists (cp | cp.port.IsC3Port ()) and
exists (cr | cr.port.IsC3port ()))

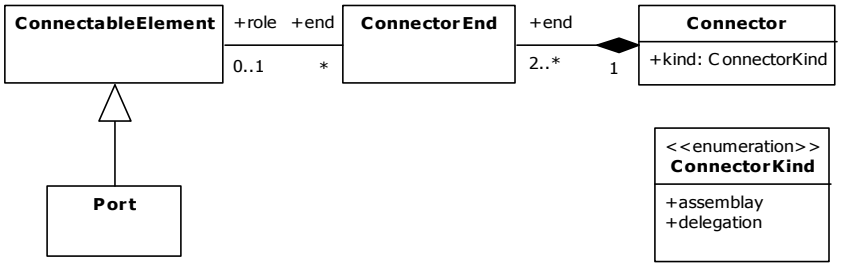


Figure 4.18- La méta-classe *Connector* dans le méta-modèle UML2.0.

4.5.4.7 Configurations de C3

La configuration de C3 est définie comme un composant d'UML2.0 stéréotypé. Elle possède zéro ou plusieurs interfaces et doit être considérée comme étant un type. <<ConfigurationC3>> hérite de *<<C3Element>>*. <<ConfigurationC3>> est un graphe de composants et de connecteurs C3. Une <<ConfigurationC3>> possède au moins un <<composantC3>> (une configuration n'est pas vide).

Stereotype << C3Configuration>> :: Component

Context UML :: InfrastructureLibrary :: Core :: Constructs :: Component
inv : self.isStereotyped ("C3Configuration") implies
self.oclIsKindOf(C3Element)
self.contents()→ forAll(el | el.stereotype = C3Component or el.stereotype = C3Connector)
self.ownedPort → forAll (i| i.IsInterfaceComposant ()) and
self.ownedPort.Size () >= 0
self.contents() → oclIsKindOf(C3component) → Size() >=1

4.5.4.8 Use de C3

Le concept d'*association* correspond au concept de *Use* de C3 dans UML2.0. Il représente le rattachement physique entre un port et un service. Use de C3 est définie seulement entre un port C3 et un service C3.

Stereotype <<Use>> :: Association

Context UML :: InfrastructureLibrary :: Core :: Constructs :: Association
inv : self.isStereotyped ("Use") implies
self.end → (exists (cp1, cp2 | cp1 <> cp2 and
cp1.port.IsPort () and cp2.port.IsService() or cp1.port.IsPort () and cp2.port.IsService())

4.6 Expérimentations

Dans cette section nous présentons deux exemples pour illustrer comment le méta-modèle C3 peut être utilisé pour décrire l'architecture des systèmes logiciels. Nous avons choisi deux applications très répandues dans la communauté des architectures logicielles : un exemple de type *Client-Serveur* et un exemple de type *Pipe-Filter*. Ensuite, nous comparons notre travail avec le langage *ArchJava* qui offre les moyens de raffiner l'architecture jusqu'au code Java. En effet, *ArchJava* intègre la description architecturale dans le langage *Java*, assure que le code est conforme à la structure architecturale, étend Java avec des dispositifs pour documenter l'architecture logicielle et supporte plusieurs styles et idiomes qui sont familiers aux programmeurs de Java.

4.6.1 Outil de modélisation

Pour réaliser nos expérimentations, nous avons choisi l'outil *UModel*© de l'éditeur Altova ©. *UModel Enterprise Edition* est une application de modélisation UML abordables, avec une interface visuelle très riche et des fonctionnalités très avancées. Il comprend de nombreuses fonctions de haute gamme permettant de manipuler les aspects les plus pratiques d'UML version 2.3.

Les profils et les stéréotypes sont utilisés pour étendre le méta-modèle UML. L'extension principale est la construction d'un stéréotype, faisant lui-même partie du profil. Les profils doivent toujours être liés à un méta-modèle de référence tel qu'UML, ils ne peuvent pas exister par eux-mêmes.

- Les profils sont des types spécifiques de paquetages, qui sont appliqués à d'autres paquetages,
- Les stéréotypes sont des méta-classes spécifiques, qui étendent des classes standards,
- Les valeurs marquées sont les valeurs d'attributs d'un stéréotype.

L'application de profil montre les profils qui ont été appliqués à un paquetage. Elle représente le type du paquetage importé qui indique qu'un profil est appliqué à un paquetage. Le profil étend le paquetage, sur lequel il a été appliqué. L'application d'un profil signifie que tous les stéréotypes qui font partie de celui-ci, sont également disponibles pour le paquetage.

L'application de profil à un paquetage architectural est indiquée par une ligne en traits pointillés avec une flèche du coté du profil appliqué, avec le mot clé « *apply* » comme le montre la Figure 4.19.

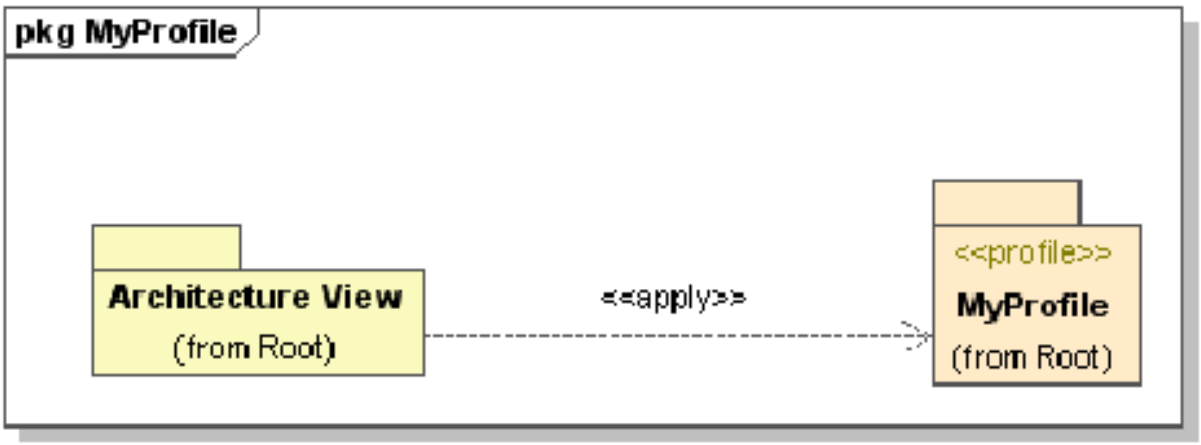


Figure 4.19- Notation graphique de l'application d'un profil à un paquetage.

Un stéréotype définit la façon dont un méta-modèle existant peut être étendu. Il représente une sorte de classe qui étend les classes par le biais d'extensions. Les stéréotypes ne peuvent être créés que dans des profils. Les stéréotypes sont affichés sous forme de classes, dans les diagrammes de classes, avec l'ajout des mots-clés « *stereotype* » au-dessus du nom de la classe.

- Les stéréotypes peuvent avoir des propriétés,
- Lorsque le stéréotype est appliqué à un élément du modèle, les valeurs de propriétés sont appelées « *valeurs marquées* ».

4.6.2 Le système Client-Serveur en C3

L'exemple Client-Serveur est très rependu au sein de la communauté d'architecture logicielle. Il est traité explicitement par la plupart des ADLs. Il décrit un système avec deux composants (Client et Serveur) qui communiquent par l'intermédiaire d'un connecteur de type RPC. L'architecture globale du système client-serveur est définie par une configuration (*CS-config*) comme le montre la Figure 4.20.

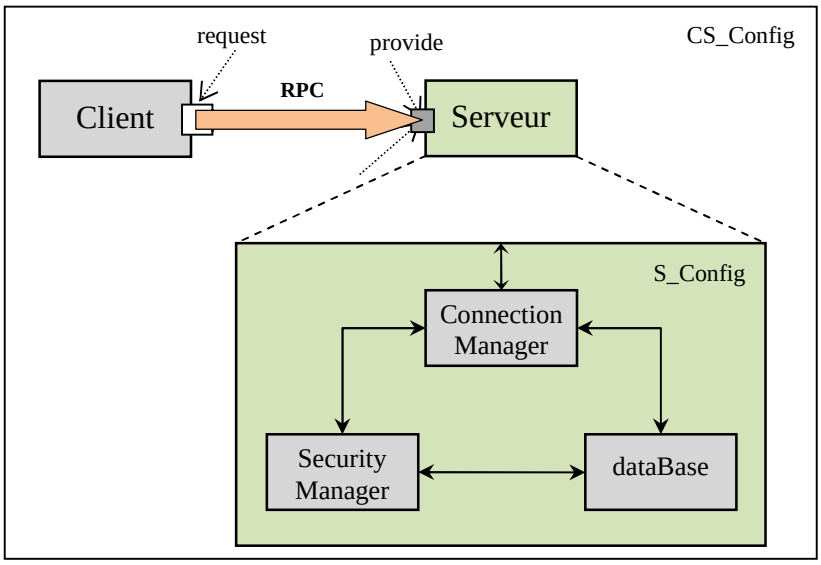


Figure 4.20- Description générale de l'exemple Client-Serveur.

4.6.2.1 Présentation de l'exemple

L'exemple est défini par une configuration (*CS_Config*) qui contient deux composants (*Client* et *Serveur*). Le composant Client possède un port requis (*request*) et un service requis (*demand-data*) et il a deux propriétés non-fonctionnelles (*data-type* et *request-rate*). Le composant Serveur possède un port fourni (*provide*) et un service fourni (*send-data*) et il a deux propriétés non-fonctionnelles (*password-needed* et *data-type*).

Le composant Serveur est défini par une configuration (*S_Config*) qui contient trois composants (*connectionManager*, *securityManager* et *dataBase*) et trois connecteurs (*SQLQuery*, *clearanceRequest* et *securityQuery*).

Le connecteur RPC agit comme un médiateur entre le composant Client et le composant Serveur, il possède un rôle fourni (*participator-1*) et un rôle requis (*participator-2*) et son type de service est *Conversion*.

4.6.2.2 Description architecturale de l'exemple Client-Serveur avec C3

C3 décrit un système comme une architecture de composants reliés par l'intermédiaire de connecteurs. Chaque composant a une interface avec des points d'interactions et des services. Les connecteurs dans C3 sont des entités de première classe qui peuvent avoir des fonctionnalités bien définies par la glu et exprimées par les interfaces. Les configurations de C3 représentent les topologies des architectures. Pour réaliser la relation de composition, C3 utilise le concept de configuration pour définir les éléments internes (c'est-à-dire, les éléments internes d'un composant composite sont groupés dans une configuration). Pour illustrer comment les concepts et les relations de C3 peuvent être utilisés pour décrire un système, nous nous appuyons sur la Figure 4.21 décrivant un système de Client-Serveur. L'exemple donne seulement une architecture (*Architecture-1*), d'autres architectures peuvent être obtenues par ré-instanciation de la configuration Client-Serveur.

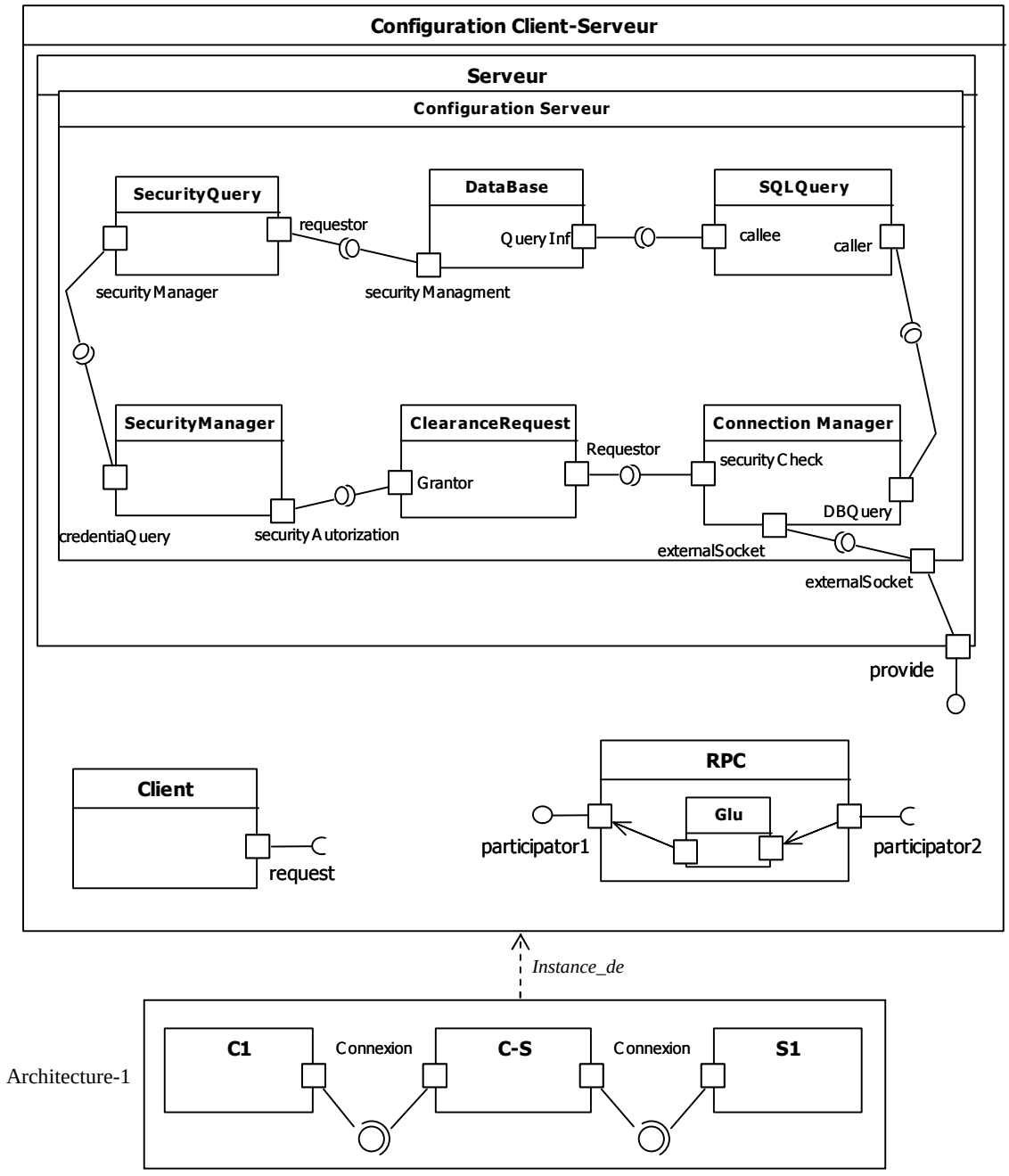


Figure 4.21- Description détaillée du système Client-Serveur (*Architecture & Application*).

4.6.2.3 Description de l'exemple Client-Serveur avec Java

La dernière étape pour obtenir l'implémentation du système Client-Serveur est le passage vers le code Java. A partir du modèle UML du système, nous passons à la phase

d'implémentation. D'abord, nous définissions les règles de passage et ensuite nous réalisons ce passage.

Le Tableau 4.5 montre les règles de passage vers le code Java, qui nous permettent d'effectuer la transition entre UML2.0 et le langage Java. L'annexe A présente un résumé de code source pour le système *Client-Serveur* selon la description de C3 et le modèle UML2.0.

Concepts C3	Concepts UML 2.0	Concepts Java	Description
Configuration	Composant	Classe	La configuration est réifiée comme une classe Java
Composant	Composant	Classe	Les composants sont réifiés grâce au concept de classe Java, qui a une sémantique forte. Une instance de composant devient alors un objet d'une classe.
Connecteur	Classe	Classe	Un connecteur est réifié au même titre qu'un composant, ce qui lui confère une sémantique explicite.
Interface	Port	Interface	Une interface Java regroupe les méthodes correspondantes aux services fournis et requis pour un composant ou un connecteur.
Port	Interface	Méthode	Les ports et les services sont définis comme des méthodes Java. Une méthode simule un service d'un composant. Elle doit appartenir à une interface pour être exportée d'une classe à une autre.
Service	Opération	Méthode	
Glu	Classe association	Classe	Une classe Java définit le concept de Glu.

Tableau 4.5- Règles de passage UML2.0 vers le code Java pour l'exemple Client-Serveur.

4.6.3 Le système Pipe-Filter en C3

Afin d'illustrer le système *Pipe-Filter*, nous avons travaillé sur une application qui se base sur le style architectural *Pipe-Filter*. Suivant ce style, des données transitent à travers le système via un connecteur (*Pipe*), puis elles sont filtrées selon les fonctions du logiciel mis en œuvre (*Filter*). Dans notre cas l'application concerne le procédé de *Capitalisation*.

4.6.3.1 Présentation de l'exemple

Le principe de fonctionnement de l'application est très simple. Il s'agit tout simplement, à partir d'un texte source d'obtenir un texte cible ayant subit la transformation demandée, c'est-à-dire avoir transformé des caractères en majuscule et d'autres en minuscules. Par exemple, prenons la chaîne de Caractères « Bonjour ceci est un texte de test » qui devient « BoNjOuR cEcI eSt Un tExTe dE TeSt ». La Figure 4.22 montre le schéma général du système.

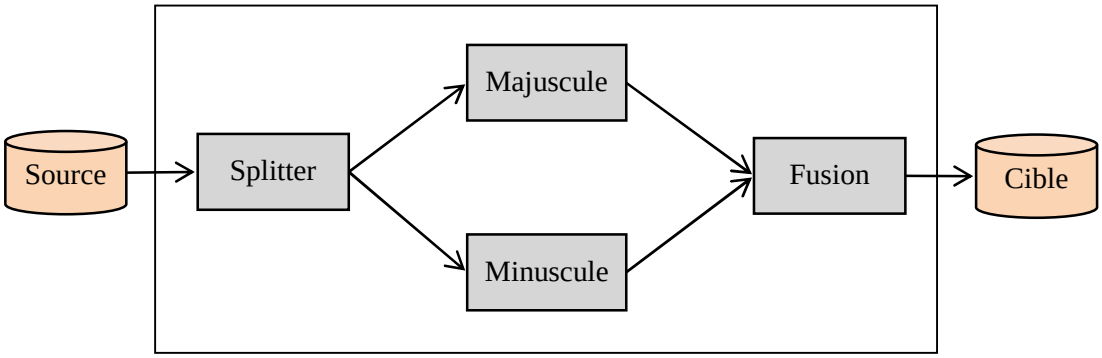


Figure 4.22- Description générale du système Capitalisation.

4.6.3.2 Description architecturale de l'application Capitalisation avec C3

L'application *Capitalisation* peut être décrite dans C3 avec :

- quatre composants (*Splitter*, *Majuscule*, *Minuscule* et *Fusion*),
- quatre connecteurs (*Pipe1*, *Pipe2*, *Pipe3* et *Pipe4*).

Ces éléments sont définis à l'intérieur d'une configuration (*Capitalisation*).

1. Le composant *Splitter* possède un port requis nommé « *In* » et deux ports fournis nommés « *toMaj* » *et* « *toMin* ».
2. Le composant *Majuscule* possède un port requis nommé « *carMin* » et un port fourni nommé « *carMaj* ».
3. Le composant *Minuscule* possède un port requis nommé « *carMaj* » et un port fourni nommé « *carMin* ».
4. Le composant *Fusion* possède deux ports requis nommés « *Majuscule* », « *Minuscule* » et un port fourni « *Cible* ».

Avec ces éléments, l'architecture de l'application Capitalisation en appliquant C3 sur UML est donnée par la Figure 4.23.

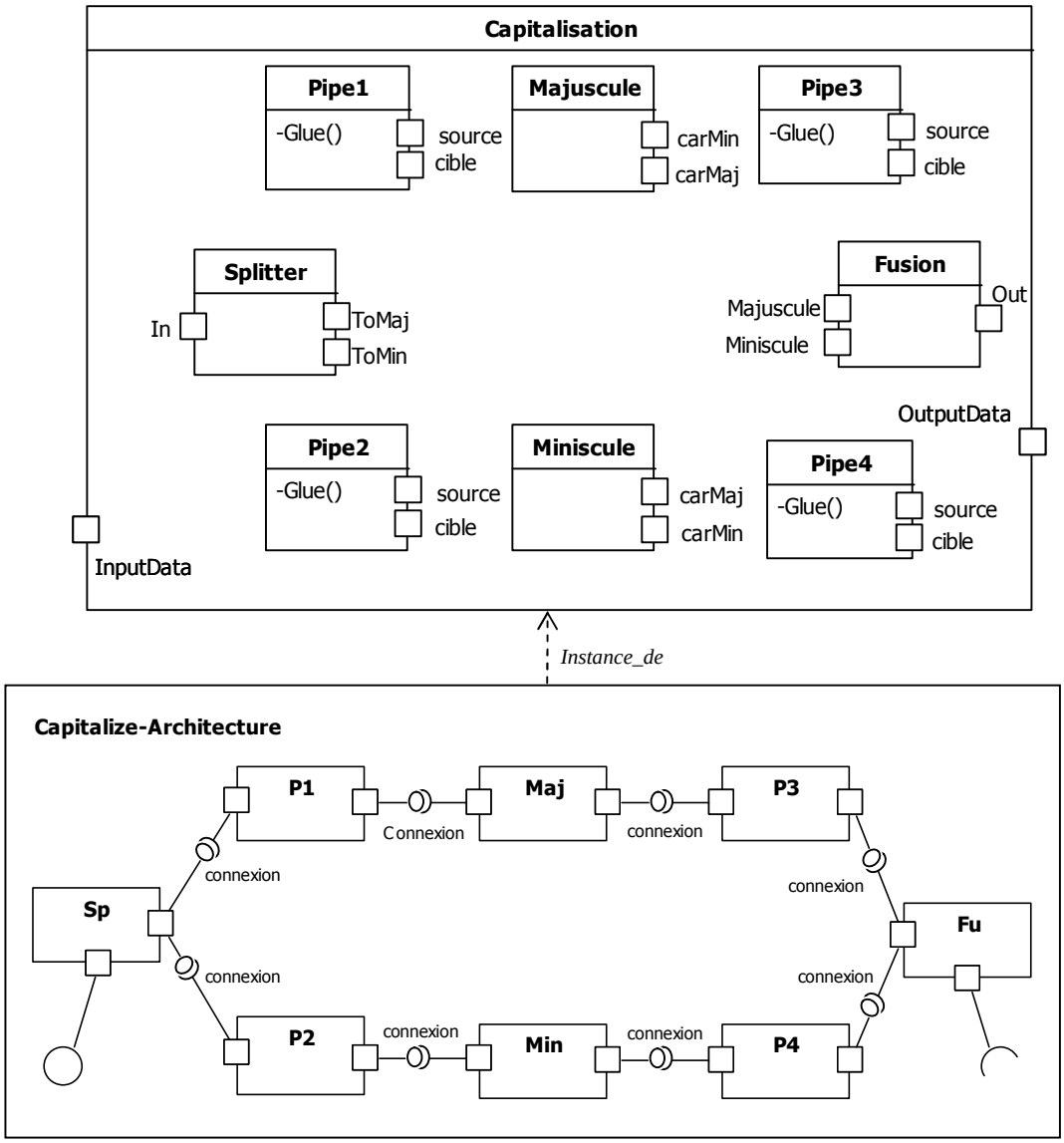


Figure 4.23- Architecture de l'application Capitalisation (*Architecture & Application*).

Pour implémenter cette application en langage Java nous avons tout d'abord conçu un diagramme en définissant les composants comme des classes et aussi nous considérons les connecteurs comme des classes à part entière.

4.6.3.3 Description de l'application Capitalisation avec le langage Java

La dernière étape pour obtenir l'implémentation de l'application Capitalisation est le passage vers le code Java. A partir du modèle UML de l'application, nous passons à la phase d'implémentation. Dans un premier temps, nous définissions les règles de passage et ensuite nous réalisons ce passage proprement dit.

Le Tableau 4.6 décrit les règles de passage vers le code Java, qui nous permettent d'effectuer la transition entre UML2.0 et le langage Java. L'annexe A présente un résumé de code source pour le système *Capitalisation* selon la description de C3 et le modèle UML2.0.

Concepts C3	Concepts UML 2.0	Concepts Java	Description
Configuration	Composant	Classe	La configuration est réifiée comme une classe Java.
Composant	Composant	Classe	Les composants sont réifiés grâce au concept de classe Java, qui a une sémantique forte. Une instance de composant devient alors un objet d'une classe.
Connecteur	Classe	Classe	Un connecteur est réifié au même titre qu'un composant, ce qui lui confère une sémantique explicite.
Port	Interface	Méthode	Les ports et les services sont définis comme des méthodes Java. Une méthode simule un service d'un composant. Elle doit appartenir à une interface pour être exportée d'une classe à une autre.
Service	Opération	Méthode	
Glu	Classe association	Classe	Une classe Java définit le concept de Glu.

Tableau 4.6- Règles de passage UML2.0 vers le code Java pour l'application Capitalisation.

4.6.4 Comparaison avec ArchJava

Ces deux exemples sont bien décrits et implémentés par Aldrich [Aldrich *et al.* 2002] en utilisant ArchJava [ArchJava 2004] que nous allons comparer avec notre travail suivant quelques caractéristiques que nous jugeons intéressantes :

- ArchJava est un langage pour les programmeurs déjà familiarisés avec le langage Java. Dans notre cas, le modèle C3 est complètement indépendant du langage d'implémentation. Pour les différents types d'intervenants (managers, programmeurs, clients, etc.), il est préférable de travailler sur une description abstraite indépendante de la plateforme d'implémentation.
- La taille du code source des applications basées sur la description de C3 est plus importante que celle du code source des applications basées sur ArchJava. Ceci s'explique, d'une part par le fait qu'avec C3 une partie du code est générée automatiquement, et d'autre part, comme ArchJava est une extension de Java [Aldrich *et al.* 2002], certaine parties du code seront réutilisées au niveau implémentation (cf. Annexe A).
- Le mélange du code relatif aux concepts de composants et de connecteurs dans ArchJava rend difficile une vision globale du système. Cependant, les concepteurs d'ArchJava ont commencé à travailler sur la séparation entre le code relatif aux connecteurs et celui relatif aux composants dans une nouvelle version du langage [Aldrich *et al.* 2003].

- Le passage d'une architecture vers le code source Java est plus facile avec ArchJava qu'avec C3. Cependant, puisque C3 est indépendant du langage d'implémentation, l'architecture peut être raffinée dans n'importe quel langage d'implémentation. Ce n'est pas le cas avec ArchJava, qui dépend complètement du langage Java.

Ainsi, si nous sommes plus intéressés par une description abstraite que par l'implémentation, il est préférable d'utiliser un modèle architectural comme C3. Par contre, si nous sommes plus intéressés par une description plus raffinée et proche du code, il est préférable d'utiliser ArchJava. Le Tableau 4.7 conclue cette comparaison selon les critères précédemment cités et qui demeurent assez significatifs.

Critères	Méta-modèle C3	ArchJava
Niveau de description	haut niveau	bas niveau
Taille du programme	grande	moyenne
Séparation des concepts	configurations, composants et connecteurs sont explicitement séparés	configurations, composants, et connecteurs sont mélangés
Raffinement	description indépendante du langage d'implémentation	Java

Tableau 4.7- Comparaison entre une description architecturale C3 et ArchJava.

4.7 Conclusion

Dans ce chapitre, nous avons proposé dans un premier temps les stratégies de projection des concepts de C3 vers UML2.0. La projection des notations architecturales vers UML est fortement encouragée en raison de la forte utilisation d'UML dans le monde industriel. Cette stratégie se décline en quatre étapes : l'instanciation de MADL par C3, la projection de MADL vers MOF, l'instanciation du MOF par UML et enfin la sélection des concepts UML les plus adéquats pour C3. Cette stratégie réduit considérablement le nombre de concepts obtenus.

Dans un second temps, nous avons montré comment on a pu réaliser le méta-modèle C3 en UML2 .0 en utilisant la technique de profil. La modélisation de C3 par UML et OCL donne une sémantique précise pour les concepts développés dans le méta-modèle C3. Ce travail représente une première étape de l'implémentation de C3. Il permet de valider notre approche sur le plan pratique. Cette validation s'effectue par une implémentation en UML2.0. Cependant, la première étape réalisée est la sélection des concepts d'UML qui représentent et préservent au mieux la sémantique des concepts architecturaux de C3. Ensuite, la modélisation de notre méta-modèle C3 est effectuée en utilisant les concepts d'UML2.0 sélectionnés précédemment. Nous avons également présenté deux expérimentations sur C3 avec une modeste comparaison avec l'ADL ArchJava.

Conclusion et perspectives

L'architecture logicielle est une problématique qui présente une forte vitalité au sein de la communauté de recherche. En témoigne le nombre de workshops et de conférences – au niveau national et international – dédiés à ce sujet qui n'a fait qu'augmenter ces dernières années. L'architecture logicielle est devenue une discipline majeure du génie logiciel, cependant, elle demeure complexe à traiter à cause de la diversité des aspects qu'elle doit prendre en considération. En effet, l'architecture doit être examinée selon plusieurs points de vue ; structurel, comportemental, fonctionnel, conceptuel et de méta-modélisation. Elle peut faire intervenir plusieurs mécanismes comme l'abstraction, la composition, le raffinement et la spécialisation. En conséquence, les pistes de recherche pour aborder cette problématique sont vastes. Face à ce constat, Medvidovic *et al.* [Medvidovic *et al.* 2007] ont tenté de mettre en exergue les grands défis que doit relever la discipline de l'architecture logicielle. Parmi lesquels, on trouve la nécessité de disposer d'une nouvelle génération d'ADLs qui donne aux concepts et aux mécanismes architecturaux un poids plus expressif et plus explicite afin de comprendre et maîtriser l'architecture des systèmes complexes. Le travail de recherche présenté dans cette thèse s'inscrit dans cette mouvance.

Dans cette thèse, nous avons circonscrit le vaste problème de description d'architectures logicielles au domaine de la description multi-hiérarchies dans les architectures à base de composants et basé toute notre recherche sur le postulat que les connecteurs et les configurations sont des concepts aussi importants et réutilisables que les composants eux mêmes. Analysons ce qui a été fait dans notre contexte.

Bilan

Les travaux décrits dans cette thèse portent sur la modélisation hiérarchique de l'architecture d'un système logiciel.

Nous avons exposé les approches de modélisation des architectures logicielles via des travaux les plus significatifs dans le domaine. La modélisation orientée objets, dite architecture logicielle à base d'objets, la modélisation orientée composants et services, dénommée architecture logicielle à base de composants si ces derniers ne sont pas répartis et dite architecture logicielle à base de services dans le cas contraire. Ce sont les deux grandes catégories d'approches qui ont émergé pour décrire les architectures logicielles, qui restent au demeurant avec de nombreux points en communs. En fait, elles sont basées sur les mêmes concepts, à savoir l'abstraction et les interactions entre les entités, cependant, chacune d'elles a ses propres avantages et également ses inconvénients.

Ainsi, nous pensons qu'une architecture logicielle à base de composants est très prometteuse pour le futur de l'architecture logicielle et participera à la réduction de la distance sémantique entre la conception et l'implémentation. Les objectifs de cette approche consistent à réduire les coûts de développement, améliorer la réutilisation des modèles, faire partager des concepts communs aux utilisateurs de systèmes et, enfin, à construire des systèmes hétérogènes à base de *composants*, de connecteurs et de configurations réutilisables qui seront disponibles sur étagères (C3OTS[1]). A travers cette étude, nous avons essayé de montrer les principaux objectifs de la modélisation des architectures logicielles à base de composants.

Par ailleurs, les langages de modélisation des architectures logicielles sont relatifs aux approches de modélisation par objets et par composants/services. Dans ce contexte, il est évident que chaque langage de modélisation est tributaire de la nature de ses entités de base qu'il manipule. Dans nos travaux de recherche, nous nous sommes intéressés très particulièrement aux entités de connecteur et de configuration. Par conséquent, nous avons abordé, avec un regard critique, la nature de la prise en charge des langages de modélisation les plus connus aux concepts de connecteur et de configuration. Pour y parvenir, nous avons dégagé les caractéristiques les plus déterminantes pour le concept de connecteur et celles pour les configurations. Ensuite, nous avons projeté les ADLs et les approches les plus connues sur chaque espace de caractéristiques. Ceci nous permet de faire une synthèse sur les points forts de chaque ADL ainsi que leurs insuffisances. Par la suite, nous avons utilisé cette synthèse pour justifier les motivations de nos travaux.

Principalement, notre travail a d'abord consisté à proposer un méta-modèle pour la description d'architecture logicielle baptisé C3 (pour : ***C****omposant,* ***C****onnecteur,* ***C****onfiguration*). C3 est composé de deux modèles ; le premier définit les concepts de base utilisés dans la description et la représentation des architectures logicielles, le second est un modèle de raisonnement défini à partir de quatre types d'hiérarchie (structurelle, fonctionnelle, conceptuelle et de méta-modélisation). Ces hiérarchies aident le concepteur à comprendre, analyser et à raisonner sur l'architecture logicielle. Chaque type d'hiérarchie représente un point de vue particulier de l'architecture.

Ensuite, nous avons mené une étude détaillée des travaux dans le domaine des architectures logicielles afin de bien définir les concepts et les éléments nécessaires pour décrire explicitement l'architecture d'un système logiciel. Suite à celle-ci, nous avons proposé une démarche pour guider le concepteur dans son processus de modélisation d'architectures logicielles, ayant pour socle le modèle MY décrivant les concepts des architectures logicielles selon trois branches : composant, connecteur et configuration. Ces branches représentent respectivement tout ce qui est lié aux calculs, aux interactions et à la structure et la topologie du système décri. Le modèle MY a quatre niveaux conceptuels : méta-méta architecture où sont décrits les méta-concepts, méta architecture où sont décrits les concepts, architecture où sont définis les types de concepts et le niveau application où sont décrites les instances des types de concepts. Enfin, ce modèle améliore la réutilisation des architectures logicielles en supportant une bibliothèque multi-hiérarchique pour chacun des niveaux conceptuels précités.

[1] *C3OTS : Component, Connector and Configuration Off-The-Shelf.*

Suite à la prolifération du nombre de langages de description d'architecture, nous avons pensé à abstraire les concepts des architectures comme les composants, les connecteurs et les configurations en utilisant le méta-modèle MADL (*Meta Architecture Description Langage*) défini pour l'architecture logicielle. Un tel méta-modèle permet, en outre, de faciliter la manipulation, la réutilisation et l'évolution des architectures logicielles. De même, qu'il permet la transformation et la comparaison entre les ADLs. Enfin, en se basant sur ce méta-modèle, nous avons décrit une stratégie de projection de concepts C3 vers UML2.0. Cette projection est fortement encouragée en raison de la popularité d'UML dans le monde industriel. La stratégie de projection est réalisée en quatre étapes : l'instanciation de MADL par C3, la projection de MADL ver le MOF, l'instanciation du MOF par UML2.0 et enfin la sélection des concepts UML2.0 les plus adéquats pour C3. Cette stratégie réduit sensiblement le nombre de concepts obtenus.

Ceci nous a permet de proposer une solution de passage entre C3 et UML2.0 via un Profil UML dont l'intérêt est largement reconnu chez la communauté scientifique [Ivers *et al.* 2004] [Oquendo 2006]. Nous avons utilisé notre profil C3-UML pour décrire deux exemples d'applications de type *Client–Serveur* et *Pipe–Filter* dont le résultat était très satisfaisant.

Ce travail n'est pas pour autant clos, des approfondissements restent à faire. Voyons ce qu'il en est.

Perspectives

Le travail de thèse a permis la mise en place d'un cadre conceptuel pour la modélisation des architectures logicielles. Il s'agit d'une approche de modélisation des architectures logicielles à hiérarchies multiples, d'une démarche d'élaboration d'une architecture logicielle et d'un modèle de projection des descriptions architecturale de C3 vers UML2.0. Cette thèse peut se prolonger vers plusieurs perspectives de recherche à moyen et à long termes, qui peuvent se décliner en sept points :

1. Description comportementale

Nous avons considéré dans le chapitre 3 uniquement les hiérarchies structurelles, fonctionnelles, conceptuelles et de méta-modélisation, cependant la description hiérarchique peut concerner aussi l'aspect comportemental des éléments architecturaux. Nous considérons que l'intégration de l'aspect comportemental dans notre travail actuel ne sera que complémentaire pour le modèle de description d'architecture que nous avons proposé.

2. Styles architecturaux

Cette seconde perspective concerne l'amélioration de l'approche C3 en définissant des contraintes et des règles spécifiques pour supporter des styles architecturaux. Comme nous l'avons déjà souligné au chapitre 1, les styles architecturaux définissent une famille de systèmes en termes de patterns d'organisation structurelle. Ils identifient l'ensemble du

vocabulaire désignant le type des entités de base (*composants* et *connecteurs*) tels que pipe, filtre, client, serveur, événement, processus, tableau noir, etc. Aussi, ils spécifient l'ensemble des règles de *configuration* qui permettent les compositions et les assemblages autorisés entre ces différents éléments. Enfin, ils mettent au point les analyses qui peuvent être réalisées sur un *système* construit selon un tel ou tel style architectural. Si nous arrivons à intégrer des styles architecturaux dans C3 nous pouvons ainsi définir de nouvelles architectures en composant différents styles (architectures hétérogènes [Mehta et Medvidovic 2003]). Les systèmes logiciels se composent généralement de différents styles. Alors, un architecte sera en mesure de comprendre les inter-relations entre ces différents styles [Le-Goaer 2009]. Dans C3, en plus des composants et des connecteurs, les configurations peuvent avoir des styles et ainsi peuvent être elles aussi composées. Si nous intégrons des styles architecturaux à C3, les architectures hétérogènes seront plus faciles à définir et à manipuler.

3. Mécanisme de raffinement et versionnement

Le but ultime de n'importe quel langage de modélisation et de conception est de produire un système exécutable [Medvidovic et Taylor 2000]. Un modèle architectural élégant et efficace n'a de valeur que s'il peut être converti en application exécutable. Dans C3, nous pouvons réaliser ceci en projetant l'architecture vers UML (particulièrement UML 2.0) et en utilisant ensuite ses outils pour obtenir le code source. Cependant, ce processus peut avoir comme conséquence beaucoup de problèmes de consistance, de vérification et de traçabilité entre l'architecture et son système exécutable. Il est donc fortement souhaitable de fournir un mécanisme de raffinement pour le modèle C3. Dans le même ordre d'idée, le versionnement peut être adressé par l'introduction d'une nouvelle relation sémantique de dérivation entre les connecteurs d'une part et les configurations d'autre part ; comme c'est le cas avec les composants. Ainsi, il sera possible de faire coexister dans la bibliothèque plusieurs versions et plusieurs raffinements d'un connecteur ou d'une configuration.

4. Validation

Cette perspective concerne la validation du méta-modèle C3 et du profil C3-UML. Certes, les deux applications *Client-Serveur* et *Pipe-Filter* nous ont apporté des éléments de réponse quant à l'application de notre approche sur des applications souvent utilisées par la communauté scientifique du domaine. Toutefois, ces expérimentations restent insuffisantes pour valider l'ensemble de nos propositions. La pratique de notre approche et de notre modèle sur des cas réels nous permettra d'avoir de plus amples retours d'expériences. Ainsi, il serait possible d'étudier de manière concrète les limites de nos propositions.

5. Description architecturale des procédés logiciels

Nous constatons que les approches existantes de description des procédés logiciels n'exploitent pas suffisamment les avancés du domaine des architectures logicielles pour la réutilisation des procédés logiciels déjà définis. Nous pensons qu'il est très intéressant alors

de développer une approche permettant la structuration et le déploiement de l'architecture des procédés logiciels. La construction d'une ontologie permet de regrouper les connaissances (expériences et savoir faire) du domaine afin de déduire une architecture globale, ensuite modéliser les connaissances inférées sous forme de configurations architecturales de procédés logiciels. Ces configurations doivent être construites à partir de composants qui représentent les phases ou les sous-phases d'un procédé, de connecteurs d'assemblages et de compositions entre les phases du procédé. Ce travail fait l'objet actuellement d'investigations au sein de notre équipe. Une thèse de doctorat sur l'extension des travaux existants (méta-modèle SPEM[2]) pour prendre en compte la description architecturale des procédés logiciels est en cours.

6. Architecture et cycle de vie des logiciels

Actuellement, la modélisation architecturale reste beaucoup plus centrée conception. Avec les notations de modélisations architecturales qui commencent à gagner en maturité, nous devons planifier pour établir une traçabilité très forte de l'architecture avec les autres activités de développement comme l'analyse des besoins, l'implémentation, les tests, la maintenance et l'évolution. Ces extensions sont nécessaires pour réaliser pleinement le rôle principal que l'architecture logiciel devra jouer dans le cycle de développement des systèmes logiciels complexes.

7. Développement d'un méta-modèle pour la composition de services

On peut s'inspirer du travail sur la modélisation des configurations explicites et des composants composites pour s'attaquer à la problématique de modélisation des services composites (*compositions de services*) qui se pose dans le domaine des architectures orientées services. Notons que les architectures à base de composants et celles à base de services ont plusieurs concepts et mécanismes en commun. Ce travail aussi, fait l'objet actuellement d'investigations au sein de notre équipe. Une thèse de doctorat dont l'objectif est l'élaboration d'un méta-modèle pour la composition de services est en cours.

[2] SPEM : *Software Process Engineering Metamodel*

Liste des publications de nos travaux

Revues internationales avec comité de lecture

1. Abdelkrim Amirat and Mourad Oussalah, "*Systematic Construction of Software Architecture Supported by Enhanced First-Class Connectors"*. **Informatica Journal** – An International Journal of Computing and Informatics, Slovenian Society Informatika, Volume 33, Issue 4, 2009, Pages 499-509, **ISSN 0350-5596**.

2. Abdelkrim Amirat and Mourad Oussalah, "*First-Class Connectors to Support Systematic Construction of Hierarchical Software Architecture*". **Journal of Object Technology** (**JOT**), ETH Swiss Federal Institute of Technology, Nov/Dec 2009, Pages 107-130, **ISSN 1660-1769.**

3. Abdelkrim Amirat and Mourad Oussalah, "*Connector Based Metamodel For Architecture Description Language*", **Journal of Computer Science** – INFOCOMP, Brazil, Volume 8, Issue 1, Pages 55-64, March, 2009, **ISSN 1807-4545.**

4. Abdelkrim Amirat, Mohamed Tayeb Laskri, and Tahar Khammaci, "*Modularization of Crosscutting Concerns in Requirements Engineering*", The International Arab Journal of Information Technology, Volume 5, Issue 2, Pages 120-125, April 2008**, ISSN 1683-3198.**

5. Abdelkrim Amirat and Mourad Oussalah, "*Four Reasoning Models for C3 MetaModel*", International Review on Computers and Software (IRECOS), Praise Worthy Prize, Volume 2, Issue 6, Italy, Pages 594-601, November 2007, **ISSN 1828-6003.**

6. Abdelkrim Amirat "*Towards a Requirements Model for Crosscutting Concerns*" Information Technology Journal (ITJ), Volume 6, Issue 3, Pages 332-337, ANSI-Journals, Pakistan 2007, **ISSN 1812-5638**.

Conférences Internationales avec comité de lecture

1. Abdelkrim Amirat and Mourad Oussalah, "*Towards an UML Profile for the Description of Software Architecture"*, International Conference on Applied Informatics (**ICAI'09**), November 15-17, 2009, Boudj Bou Arréridj, Algeria, Pages 226-232, **ISBN 978-9947-0-2763-9**.

2. Abdelkrim Amirat and Mourad Oussalah, "*Reusable Connectors in Component-Based Software Architecture*", the ninth international symposium on programming and systems, (**ISPS 2009**), May 25-27, 2009, Algiers, Algeria, Pages 28-35.

3. Abdelkrim Amirat and Mourad Oussalah "*C3: A Metamodel for Architecture Description Language Based on First-Order Connector Types*", 11th International Conference on Enterprise Information Systems (**ICEIS 2009**), May 6 - 10, 2009, Milan, Italy, Pages 76-81, **ISBN 978-3-642-01346-1.**

4. Abdelkrim Amirat and Mourad Oussalah, "*Representation and Reasoning Models for C3 Architecture Description Language*", Proceedings of the Tenth International Conference on Enterprise Information Systems (**ICEIS'08)**, June 12-16, 2008, Barcelona, Spain, Pages 207-212, **ISBN 978-989-8111-38-8**.

5. Abdelkrim Amirat and Mourad Oussalah "*Structural and Behavioural Composition Aspects in Component-Based Architectures*", In Proceedings of the 1st International Conference on Web and Information Technologies (**ICWIT '08**), 29-30 Juin 2008, Sidi Bel Abbes, Algérie, **ISBN: 978-1-60566-046-2.**

6. Abdelkrim Amirat and Mourad Oussalah "*Enhanced Connectors to Support Hierarchical Dependencies in Software Architecture*", In Proceedings of the 8th international conference on New Technologies in Distributed Systems, (**NOTERE 2008**), June 23-27, ACM Edition 2008, Lyon, France, Pages 252-261, **ISBN: 978-1-59593-937-1.**

7. Abdelkrim Amirat and Mourad Oussalah " *Multi Hierarchies for Component-Based Architecture*", 6èmes Edition du Séminaire National en Informatique (**SNIB'08**), 06-08 Mai 2008, Biskra, Algérie, Pages 11-20.

8. Abdelkrim Amirat and Mourad Oussalah "*C3 : Un Métamodèle pour la Description Hiérarchique des Architectures Logicielles*", 10th Maghrebian Conference on Information Technologies (**MCSEAI'08**), April 28 - 30, 2008, Oran, Algeria, Pages 148-155.

9. Abdelkrim Amirat and Mourad Oussalah "*Interaction-Based Mechanisms to Model Hierarchical Architectures*", In Proceedings of the 23th International Conference on Computers and Their Applications (**CATA-2008**) , April 2008, Cancun, Mexico, Pages 287-292, **Editor: T. Philip, ISBN: 978-1-880843-66-6.**

10. Abdelkrim Amirat and Mourad Oussalah "*Hierarchical Model to Develop Component-Based Systems*", In Proceedings of the 15th IEEE International Conference on Engineering of Computer-Based Systems (**ECBS'08**, April 2008), University of Ulster at Belfast, Northern Ireland, UK, Pages 337-345, **ISBN: 0-7695-3141-5.**

11. Abdelkrim Amirat, Mourad Oussalah, and Tahar Khammaci, "*Towards an Approach for Building Reliable Architectures*", In Proceeding of IEEE International Conference on Information Reuse and Integration (**IEEE IRI'07**), August 2007, Las Vegas, Nevada, USA, Pages 467-472, **ISBN: 1-4244-1500-4.**

12. Mourad Oussalah, Abdelkrim Amirat, and Tahar Khammaci, "*Software Architecture Based Connection Manager*", In Proceedings of Software Engineering and Data Engineering (**SEDE'07)**, July 2007, Las Vegas, Nevada, USA, Pages 194-199, **ISBN 978-1-880843-63-5.**

13. Abdelkrim Amirat, Adel Smeda and Olivier Le-Goaer, "*Integrating Aspects in the COSA Model*", International Symposium on Programming and Systems", (**ISPS'07**), May 7-9, 2007, Alger, Algérie, Pages 79-87, **ISSN 1112-5853.**

14. Abdelkrim Amirat, Mohamed Tayeb Laskri, and Tahar Khammaci, "*Modularization of Crosscutting Concerns in Requirements Engineering*", in Proceeding of Maghrebian conference on Software Engineering and Artificial Intelligence (**MCSEAI'06**), Poster paper, December 2006, Agadir Morocco.

15. Abdelkrim Amirat, Mohamed Tayeb Laskri, and Djamel Meslati, "*Identification and Separation of Crosscutting Concerns: A Case Study*" Proceeding of the 8th African Conference on Research in Computer Science (**CARI'06**), November 6-9, 2006, Cotonou, Benin, Pages 43-50, **ISBN 2-7851-1291 9**.

Bibliographie

[ACCORD 2002] Projet ACCORD, Etat de l'Art sur les Langages de Description d'Architecture (ADLs), *Rapport technique*, INRIA, France, 2002.

[Allen 1997] Allen R., A Formal Approach to Software Architecture, *PhD Thesis*, Carnegie Mellon University, *CMU Technical Report*, CMU-CS-97-144, May 1997.

[Allen et Garlan 1997] Allen R., Garlan D., A Formal Basis for Architectural Connection, *ACM Trans. Software Eng. and Methodology*, vol. 6, no. 3, pp. 213-249, July 1997.

[Allen *et al.* 1998] Allen R., Garlan D., Ivers J., Formal Modeling and Analysis of the HLA Component Integration Standard, *Proceedings of the Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 70-79, Lake Buena Vista, November 1998.

[Aldrich *et al.* 2002] Aldrich J., Chambers C., Notkin D., ArchJava: Connecting Software Architecture to Implementation, *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, Orlando, USA, 2002.

[Aldrich *et al.* 2003] Aldrich J., Sazawal V., Chambers C., Notkin D., Language Support for Connector Abstractions, *Proceedings of the 2003 European Conference on Object-Oriented Programming (ECOOP 2003)*, Darmstadt, Germany, 2003.

[Amirat *et al.* 2007] Amirat A., Oussalah M., Khammaci T., Towards an Approach for Building Reliable Architectures, *Proceeding of IEEE International Conference on Information Reuse and Integration (IEEE IRI'07)*, Las Vegas, Nevada, USA, pp. 467-472, August 2007.

[Amirat et Oussalah 2008] Amirat A., Oussalah M., Representation and Reasoning Models for C3 Architecture Description Language, *Proceedings of the Tenth International Conference on Enterprise Information Systems (ICEIS'08)*, pp. 207-212, Barcelona, Spain, June 12-16, 2008.

[Amirat et Oussalah 2009] Amirat A., Oussalah M., Systematic Construction of Software Architecture Supported by Enhanced First-Class Connectors, *Informatica Journal – An International Journal of Computing and Informatics*, Slovenian Society Informatika, pp. 499-509, Volume 33(4), 2009.

[Angele *et al.* 1998] Angele J., Fensel D., Landes D., Studer R., Developing Knowledge Based Systems with MIKE, *Journal of Automated Software Engineering*, Volume 5(4), October 1998.

[ArchJava 2004] *ArchJava homepage*, disponible sur http://archjava.fluid.cs.cmu.edu, 2004.

[Barais et Duchien 2004] Barais O., Duchien L., SafArch : Maîtriser l'Evolution d'une Architecture Logicielle, In *Langages, Modèles et Objets*, Journées Composants, LMO 2004-JC 2004. Lille, France, Hermès Sciences, pp. 103-116.

[Barbier *et al.* 2004] Barbier F., Cauvet C., Oussalah M., Rieu D., Souveyet C., Concepts Clés et Techniques de Réutilisation dans l'Ingénierie des Systèmes d'Information, *Ingénierie des Composants dans les Systèmes d'Information*, M. Oussalah et D. Rieu (réd.), Hermes Science Publications, 2004.

[Bass *et al.* 1998] Bass L., Clements P., Kazman R., *Software Architecture in Practice*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[Benjamins 1995] Benjamins V.R., Problem solving methods for diagnosis and their role in knowledge acquisition, *International Journal of Expert Systems: Research and Application*, Volume 8(2), pp. 93-120, 1995.

[Binns *et al.* 1996] Binns P., Englehart M., Jackson, M. Vestal S., Domain-specific software architectures for guidance, Navigation and Control, *International Journal of Software Engineering and Knowledge Engineering,* Volume 6(2), pp. 201-227, June 1996.

[Blanc 2005] Blanc X., *MDA en Action: Ingénierie logicielle guidée par les modèles,* Eyrolles, 2005.

[Booch 1994] Booch G., *Object-Oriented Analysis and Design with Applications*, The Benjamin/Cummings Publishing Company Inc., Redwood City, California, 1994.

[Booch *et al.* 1998] Booch G., Rumbaugh J., Jacobson I., *The Unified Modeling Language User Guide,* Addison-Wesley Publishing, Reading, Massachusetts, 1998.

[Booch *et al.* 1999] Booch G., Garlan, D., Iyengar S., Kobryn C., Stavridou V., Is UML an Architectural Description Language, *OOPSLA,*1999.

[Booch *et al.* 2005] Booch G., Rumbaugh J., Jacobson I., *The Unified Modeling Language User Guide*, 2nd edition. Addison-Wesley Object Technology Series, Reading, Massachusetts, Addison-Wesley Professional Reading, Massachusetts, 2005.

[Breuker et Van de Velde 1994] Breuker J., Van de Velde W. e., The Common KADS Library*, Technical report*, University of Amsterdam and Free University of Brussels, 1994.

[Brinkkemper *et al.* 1995] Brinkkemper S., Hong S., Bulthuis Avan den Goor G., *Object-Oriented Analysis and Design Methods: A Comparative Review*, disponible sur http://elex.amu.edu.pl/languages/oodoc/oo.html.

[Bruneton *et al.* 2002] Bruneton E., Coupaye T., Stefani J.B., Recursive and Dynamic Software Composition with Sharing, *Workshop on Component-Oriented Programming (WCOP)* at ECOOP'02, June 2002.

[Bruneton *et al.* 2003] Bruneton E., Coupaye T., Stefani J.B., Specification of the Fractal Component Model, *Technical Report.* http://fractal.objectweb.org, 2003.

[Bruneton 2004] Bruneton E., Fractal ADL tutorial. *Technical Report*, France Télécom R&D, http ://fractal.objectweb.org/tutorials/adl/, 2004.

[Bruneton *et al.*, 2004] Bruneton E., Coupaye T., Stefani J.B., *The Fractal Component Model*, Version 2.0-3, Février 2004.

[Budinsky *et al.* 2008] Budinsky F., Merks E., Steinberg D., *Eclipse Modeling Framework* 2.0, (2nd edition), Addition-Wesley Professional, 2008.

[Buxton et Randell 1970] Buxton J.N., Randell B., Software Engineering Techniques, *Report on a conference sponsored by the NATO Science Committee,* Rome, Italy, October 1969, NATO Science Committee, pp. 12. Editors: J. N.Buxton and B. Randell, 1970.

[CCM 2002] Object Management Group, *Corba Components*, version 3.0.

[Cox 1986] Cox B.J., *Object-Oriented Programming – an Evolutionary Approach*, Addison-Wesley Publishing, Reading, Massachusetts, 1986.

[Clements et al. 2003] Clements P., Garlan D, Little R, Nord R, Stafford J., Documenting Software Architectures: Views and Beyond, *Proceeding of the 25th International Conference on Software Engineering, ICSE 2003,* Washington DC, USA, pp.740-741.

[Dashofy *et al.* 2005**]** Dashofy E., Hoek A.v.d., Taylor R.N., A comprehensive Approach for the Development of XML-based Software Architecture Description Languages, *Transactions on Software Engineering Methodology (TOSEM)*, Volume14 (2), pp.199-245, 2005.

[Dashofy *et al.* 2007] Dashofy E., Asuncion H., Hendrickson S., Suryanarayana G., Georgas J., Taylor R., ArchStudio 4: An Architecture-Based Meta-modeling Environment, *International* Conference *on Software Engineering (ICSE 2007)*, pp. 67-68, 2007.

[DeMichiel 2003] DeMichiel L. G., *Entreprise JavaBeans Specification*, version 2.1. Sun Microsystems, Novembre, 2003.

[DeRemer et Kron 1975] DeRemer F., Kron H., Programming-in-the large versus programming-in-the-small, In *Proceedings of the International Conference on Reliable Software*, New York, USA, 1975, ACM, pp. 114-121.

[Dijkstra 1968] Dijkstra E., The Structure of The-Multiprogramming System, *Communications of the ACM*, Volume 11, Number 5, pp. 341-346, May 1968.

[Eclipse 2007] Eclipse Foundation, *Eclipse*, <http://www.eclipse.org/>, HTML, 2007.

[EJB 2003] Sun Microsystems, Inc., *Enterprise Java Beans*, Version 2.1, November 2003.

[Engels et Gregor 2000] Engels G., Groenwegen L., Object-Oriented Modeling: a Roadmap, *The Future of Software Engineering*, A. Finkelstein (re-ed.), ACM press, pp. 103-116, 2000.

[Enrique et Martinez 2003] Enrique J., Martinez P., Heavyweight extensions to the UML metamodel to describe the C3 architectural style, *ACM SIGSOFT Software Engineering Notes*, volume 28(3), May 2003.

[Feiler *et al.* 2003] Feiler P.H., Lewis B., Vestal S.: The SAE avionics architecture description language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. *Proceedings of the RTAS 2003 Workshop on Model-Driven Embedded Systems,* Washington, DC., May, 2003.

[Garlan et Shaw 1993] Garlan D., Shaw M., An introduction to software architecture, *Advances in Software Engineering and Knowledge Engineering*, Singapore, V. Ambriola and G. Tortora (réd.),WorldScientic Publishing Company, Singapore, pp. 1-39, 1993.

[Garlan *et al.* 1994] Garlan D., Allen R., Ockerbloom J., Exploiting Style in Architectural Design Environments, *Proceeding of SIGSOFT'94*, Foundations of Software Engineering, pp. 175-188, December 1994.

[Garlan et Perry 1995] Garlan D., Perry D., Introduction to the Special Issue on Software Architecture, *IEEE Transactions on Software Engineering*, vol. 21(4), April 1995.

[Garlan 1995] Garlan D., *An Introduction to the Aesop System*, July 1995. http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesop-overview.ps.

[Garlan *et al.* 1997] Garlan, D., Monroe, R.T., Wile D., ACME: An architecture description interchange language, *Proceedings of the CASCON'97,* pp. 169-183, IBM Center for advanced studies, Toronto, Canada, 1997.

[Garlan 2000] Garlan D., Software architecture: a roadmap. *ICSE '00: Proceedings of the Conference on the Future of Software Engineering*, ACM Press, pp. 91–101, USA, 2000.

[Garlan *et al.* 2000] Garlan D., Monroe R., Wile D.: Acme, Architectural Description of Component-Based Systems, Foundations of Component-Based Systems, *Cambridge University Press*, 2000, pp. 47-68.

[Garlan *et al.* 2002] Garlan D., Cheng S.W., Kompanek A., Reconciling the Needs of Architectural Description with Object-Modeling Notations, *Science of Computer Programming*, vol. 44 , Issue 1, pp. 23-49, 2002.

[Gorlick et Razouk 1991] Gorlick M., Razouk R.R., Using Weaves for Software Construction and Analysis, *Proceeding of 13th International Conference on Software Engineering (ICSE13)*, pp. 23-34, May 1991.

[Goulão et Abreu 2003] Goulão M., Abreu F.B., Bridging the gap between Acme and UML 2.0 for CBD, *Workshop at ESEC/FSE'03*-Septembre 1-2, 2003.

[Heineman et Councill 2001] G. Heineman et W. Councill, *Component-Based Software Engineering – Putting the Pieces Together*, Addison-Wesley Publishing, Reading, Massachusetts, 2001.

[Hilliard 1999] Hilliard, R., Using the UML for Architectural Description, *Proceedings of the UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference.* Fort Collins, CO, USA, 1999.

[Hofmeister *et al.* 1999] Hofmeister C., Nord R.L., Soni D., Describing Software Architecture with UML, *Proceedings of the First IFIP Working Conference on Software Architecture,* San Antonio, TX, February, 1999.

[IEEE 2000] IEEE Architecture Working Group: *IEEE Standard* 1471-*2000*, *Recommended Practice for Architectural Description of Software-Intensive Syste*ms, Tech. Report, IEEE, 2000.

[Ivers *et al.* 2004] Ivers J., Clements P., Garlan D., Nord R., Schmerl B., Silva J.R.O., Documenting component and connector views with UML 2.0, *Technical report CMU/SEI-2004*-TR-008, April 2004.

[Jacobson *et al.* 1992] Jacobson I., Christerson M., Jonsson P., Övergaard G., *Object-Oriented Software Engineering*, Addison-Wesley Publishing, Reading, Massachusetts, 1992.

[Jouault et Bézivin 2006] Jouault*,* F*.,* Bézivin J., *KM3: a DSL for Metamodel Specification*, FMOODS 2006, Bologna, Italy, 14-16 June 2006.

[Kandé et Strohmeier 2000] Kandé M. M., Strohmeier A., Towards a UML Profile for Software architecture description, *Proceedings of UML'00*, Third International Conference, York, UK, October 2-6, 2000.

[Kazman 2001] Kazman R., *Software Architecture*, Handbook of Software Engineering and Knowledge Engineering, Volume 1, Fundamentals, SK. Chang (réd.), World Scientific Publishing Co., Singapore, 2001.

[Kenney 1995] Kenney J. J., *Executable formal models of distributed transaction systems based on event processing*, PhD thesis, Stanford University, December 1995.

[Khare *et al.* 2001] Khare R., Guntersdorfer M., Oreizy P., Medvidovic N., Taylor R.N, xADL: Enabling Architecture-Centric Tool Integration with XML, *Proceedings of the 34th Hawaii International Conference on System Sciences (HICSS-34), Software mini-track.* Maui, Hawaii, January 3-6, 2001.

[Lau *et al.* 2007] Lau K.K., Ling L., Ukis V., and Elizondo P.V., Composite Connectors for Composing Software Components, *Lecture Notes in Computer Science*, Volume 4829, pp. 266-280, 2007.

[Le-Goaer 2009] Le-Goaer O., *Styles d'évolution dans les architectures logicielles*, Thèse de Doctorat, Laboratoire LINA, Université de Nantes, 2009.

[Luckham *et al.* 1995] Luckham D.C., Kenney J.J., Augustin L.M., Vera J., Bryan D., Mann W., Specification and Analysis of System Architecture Using Rapide, *IEEE Transactions on Software Engineering*, volume 21, number 4, pp. 336-355, April 1995.

[Luckham et Vera 1995] Luckham D.C., Vera J., An Event-Based Architecture Definition Language, *IEEE Transaction on Software Engineering*, volume 21(9), pp. 717-734, Sept. 1995.

[Magee *et al.* 1994] Magee J., Dulay N., Kramer J., A constructive development environment for parallel and distributed programs, *IEE/IOP/BCS Distributed Systems Engineering Journal*, volume 1, number 5, September 1994.

[Magee *et al.* 1995] Magee J., Dulay N., Eisenbach S., Kramer J., Specifying Distributed Software Architectures, *Proc. Fifth European Software Eng. Conf. (ESEC '95)*, Sept. 1995.

[Martin 1993] Martin J., *Principles of Object-Oriented Analysis and Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[Matevska-Mayer *et al.* 2004] Matevska-Meyer J., W. Hasselbring W., Reussner R., Software Architecture Description Supporting Component Deployment and System Runtime Reconfiguration, *Proceedings of Workshop on Component-Oriented Programming WCOP 2004*, Oslo, Norway, June 2004.

[Matougui et Beugnard 2005] Matougui S. et Beugnard A., How to Implement Software Connectors? A Reusable, Abstract and Adaptable Connector, *Proceeding of Distributed Applications and Interoperable Systems*, DAIS 2005, pp. 83-94.

[McIlroy 1968] McIlroy D., Mass-produced Software Components, *Proceedings of the 1st International Conference on Software Engineering*, Garmisch Pattenkirschen, Germany, October 1968.

[Medvidovic *et al.* 1996] Medvidovic N., Oreizy P., Robbins J.E., Taylor R.N., Using Object-Oriented Typing to Support Architectural Design in the C2 Style, *Proc. ACM SIGSOFT '96: Fourth Symp. Foundations Software of Eng. (FSE4)*, pp. 24-32, Oct. 1996.

[Medvidovic *et al.* 1999] Medvidovic N., Rosenblum D.S., Taylor R.N., A Language and Environment for Architecture-Based Software Development and Evolution, *Proceeding 21st International Conference on Software Engineering (ICSE'99)*, pp. 44-53, May 1999.

[Medvidovic et Taylor 2000] Medvidovic N., Taylor R., A Classification and Comparison Framework for Software Architecture Description Languages, *IEEE Transactions on Software Engineering*, 26(1) pp.70–93, January 2000.

[Medvidovic *et al.* 2002] Medvidovic N., Rosenblum D.S., Redmiles D., Robbins J.E., Modeling Software Architectures in the Unified Modeling Language, *ACM Transactions on Software Engineering and Methodology (TOSEM),* 11(1), pp. 2-57, January 2002.

[Medvidovic *et al.* 2007] Medvidovic N., Dashofy E., Taylor R.N.: Moving Architectural Description from Under the Technology Lamppost, *Information and Software Technology,* volume 49(1), pp. 12-31, January, 2007.

[Mehta et Medvidovic 2003] Mehta N., Medvidovic N., Composing Architectural Styles from Architectural Primitives, *Proceedings of the Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering. ESEC/FSE 2003*, Helsinki, Finland, September 2003.

[Menzies 2002] Menzie T., *SE/KE Reuse Research: Common Themes and Empirical Results*, vol. 2, World Scientific Publishing Co., 2002, Chapter in Handbook of Software Engineering and Knowledge Engineering. 2002.

[Meyer 1992] Meyer B., *Applying Design by Contract*, in *Computer (IEEE)*, volume 25, issue 10, pp. 40-51, October 1992.

[Meyer 2000] Meyer B., *Conception et programmation orientées objet*, Eryolles 2000.

[Milligan 2000] Milligan M.K.J., Implementing COTS open systems technology on AWACS, *CrossTalk: The Journal of Defense Software Engineering*, September 2000.

[Minsky *et al.* 1968] Minsky M. L., Matter, Minds, and Models, *Semantic Information Processing,* Marvin L. Minsky (ed.), MIT Press, (1968).

[Moriconi *et al.* 1995] Moriconi M., Qian X., Riemenschneider R.A., Correct Architecture Refinement, *IEEE Trans. Software Engineering*, vol. 21, no. 4, pp. 356-372, April 1995.

[Moriconi *et al.* 1997] Moriconi M., Riemenschneider R.A., Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies, *Technical Report SRI-CSL-97-01, SRI International*, March 1997.

[Motta 2000] Motta E., The Knowledge Modeling Paradigm in Knowledge Engineering, *Handbook of Software Engineering and Knowledge Engineering*, Volume 1 : Fundamentals, SK. Chang (réd.), World Scientific Publishing Co., Singapore, pp. 589-614, 2000.

[Muller *et al.* 2005] Muller P.-A., Fleurey F., Jézéquel J.-M., Weaving executability into object-oriented meta-languages, In *Model Driven Engineering Languages and Systems*, pp 264-278, LNCS, 2005.

[Oquendo 2004] Oquendo F., π-ADL: An Architecture Description Language based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures, *ACM Software Engineering Notes*, Volume 29, Issue 3 (May 2004).

[Oquendo 2006] Oquendo F., Formally Modelling Software Architectures with the UML 2.0 Profile for JI–ADL, *ACM SIGSOFT Engineering Notes*, Vol.31, No.1, January 2006.

[OMG 2002a] Object Management Group (OMG), *Meta Object Facility (MOF) Specification*, formal/2002-04-03, April 2002.

[OMG 2002b] Object Management Group OMG. *CORBA Component Model*, v3.0, formal/2002-06-65. Object Management Group OMG, Juin 2002.

[OMG 2003a] OMG, *MDA Guide*, version 1.0.1, Document Number : omg/2003-06-01. OMG document, June 2003.

[OMG 2003b] OMG (Object Management Group), *Unified Modeling Language (UML) Specification 1.5*, March 2003.

[OMG 2003c] OMG (Object Management Group), *UML2.0 OCL specification, Technical report*, Object Management Group, 2003, ptc/03-10-14.

[OMG 2005] UML2.0 Superstructure Specification, *Final Adopted Specification*, July2005.

[OMG 2007] Object Management Group. *OMG SysML Specification Coversheet,* Report, pp. 262, March 28, 2007. <http://www.omg.org/cgi-bin/apps/doc?ptc/07-02-04.pdf>.

[Ommering *et al.* 2000] Ommering R.v., Linden F.v.d., Kramer J., Magee J., The Koala Component Model for Consumer Electronics Software, *IEEE Computer,* volume 33(3), pp.78-85, March, 2000.

[Ommering 2002**]** Ommering R.v., Building product populations with software components, *22rd International Conference on Software Engineering (ICSE'2002)*, pp. 255–265, May 2002.

[Oussalah 2002] Oussalah M., Component-Oriented KBS, *SEKE'02*, pp.73-76, 2002.

[Oussalah *et al.* 2004] Oussalah M., Smeda A., Khammaci T., An Explicit Definition of Connectors for Component-Based Software Architecture, *Proceedings of IEEE International Conference on the Engineering of Computer Based Systems (ECBS 2004)*, Brno, Czech Republic, 2004.

[Oussalah *et al.* 2005] Oussalah M., Khammaci T., Smeda A., Les composants : Définitions et Concepts de Base, dans *Ingénierie des Composants : Concepts, Techniques et Outils*, Oussalah M., (réd.), Vuibert Informatique, Paris, 2005.

[Oussalah *et al.* 2007] Oussalah M., Amirat A., Khammaci T., "Software Architecture Based Connection Manager", In *Proceedings of Software Engineering and Data Engineering (SEDE'07)*, Las Vegas, Nevada, USA, Pages 194-199, July 2007.

[Parnas 1972] Parnas D., On the Criteria to be Used in Decomposing Systems into Modules, *Communications of the ACM*, 12(12), pp. 1053-1058, December, 1972.

[Perry et Wolf 1992] Perry D.E., Wolf A.L., Foundations for the Study of Software Architectures, SIGSOFT Software Engineering Notes, vol. 17, no. 4, pp. 40-52, Oct. 1992.

[Pfister et Szyperski 1996] Pfister C., Szyperski C., Why objects are not enough. *In Proceeding of the First International Component Users Conference (CUC'96)*, 1996.

[Plásil *et al.* 1998] Plasil F., Balek D., Janecek R., SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, *Proceedings of ICCDS 98*, IEEE CS Press, Annapolis, Maryland, USA, 1998.

[Prieto-Diaz et Neighbors 1989] Prieto-Diaz R., Neighbors J. M., Module Interconnection Languages, *Journal of Systems and Software*, vol. 6, no. 4, pp. 307-334, October 1989.

[Pinto *et al.* 2005] Pinto M., Fluentes L., Troya M., A Dynamic Component and Aspect-Oriented Platform, *The Computer Journal*, vol.48, n. 4, pp. 401-420, 2005.

[Robbins *et al.* 1998] Robbins J., Medvidovic N., Redmiles D., Rosenblum D., Integrating Architecture Description Languages with a Standard Design Method, In *Proceedings of the 20th Intr. Conf. on Soft. Eng. (ICSE),* pp. 209-218, ACM Press. Kyoto, Japan, April, 1998.

[Roh *et al.* 2004] Roh S., Kim K., Jeon T., Architecture Modeling Language based on UML2.0, Software Engineering Conference, APSEC'04, *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, 2004.

[Rumbaugh *et al.* 1991] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Shadbolt *et al.* 1993] Shadbolt N., Motta E., Rouge A., Constructing Knowledge-Based Systems, *IEEE Software*, volume 10 number 6, pp.34-38, November 1993.

[Shaw *et al.* 1995] Shaw M., DeLine R., Klein D.V., Ross T.L., Young D.M., Zelesnik G., Abstractions for Software Architecture and Tools to Support Them, *IEEE Transaction on Software Engineering*, vol. 21, no. 4, pp. 314-335, April 1995.

[Shaw *et al.* 1996] Shaw M., DeLine R., Zelesnik G., Abstractions and Implementations for Architectural Connections, in *Proceeding of Third International Conference on Configurable Distributed Systems*, May 1996.

[Shaw et Garlan 1996] Shaw M., Garlan D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Upper Saddle River, N.J., 1996.

[Smeda *et al.* 2005] Smeda A., Oussalah M., Khammaci, T., MADL: Meta Architecture Description Language, *Third ACIS Int'l Conference on Software Engineering Research, Management and Applications (SERA'05)*, pp. 152-159, 2005.

[Smeda 2006] Smeda A., *Contribution à l'élaboration d'une méta-modélisation de description d'architecture logicielle*, Thèse de doctorat, Université de Nantes, 2006.

[Smeda *et al.* 2008] Smeda A., Oussalah M., Khammaci T., My Architecture: a Knowledge Representation Meta-Model for Software Architecture, *International Journal of Software Engineering and Knowledge Engineering*, 18(7), pp.877-894, 2008.

[Szyperski 2002] Szyperski C, *Component Software - Beyond Object-Oriented Programming,* Addison-Wesley, November 2002.

[Taylor *et al.* 1996] Taylor R. N., Medvidovic N., Anderson K. M., Jr. E. J. W., Robbins J. E., Nies K. A., Oreizy P. Dubrow D. L., A component and message-based architectural style for GUI software, *Software Engineering*, volume 22(6), pp. 390-406, 1996.

[Taylor *et al.* 2009] Taylor R.N., Medvidovic N., Dashofy E., *Software Architecture: Foundations, Theory, and Practice*, Wiley-Blackwell, 2009.

[Tracz 1993] Tracz W. J., Parameterized Programming in *LILEANNA*, In *Proceedings of ACM Symposium on, Applied Computing* SAC'93, pp. 77-86, 1993.

[Vestal 1996] Vestal S., MetaH Programmer's Manual, Version 1.09, *Technical Report*, Honeywell Technology Center, April 1996.

[Warmer et Kleppe 2003] Warmer J., Kleppe A., *The Object Constraint Language*, Addison-Wesley, August 2003.

[Wile 1999] Wile D., AML: An Architecture Meta Language, *Proceedings 14th International Conference on Automated Software Engineering*, Cocoa Beach, FL, October 1999,

[Wirfs-Brock *et al.* 1990] Wirfs-Brock R., Wilkerson B., Wiener L., *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[Zarras *et al.* 2001] Zarras A., Issarny V., Kloukinas C., Nguyen V. K., Towards a Base UML Profile for Architecture Description, *INRIA Rocquencourt*, 2001.

[Zhang *et al.* 2010] Zhang H.(Y.), Urtado C., and Vauttier S., Architecture-centric component-based development needs a three-level ADL, *4th European Conference on Software Architecture, ECSA2010*, Copenhagen, Denmark.

ANNEXE A

Réalisation des applications Client-Serveur et Pipe-Filter

A.1 Application Client-Serveur

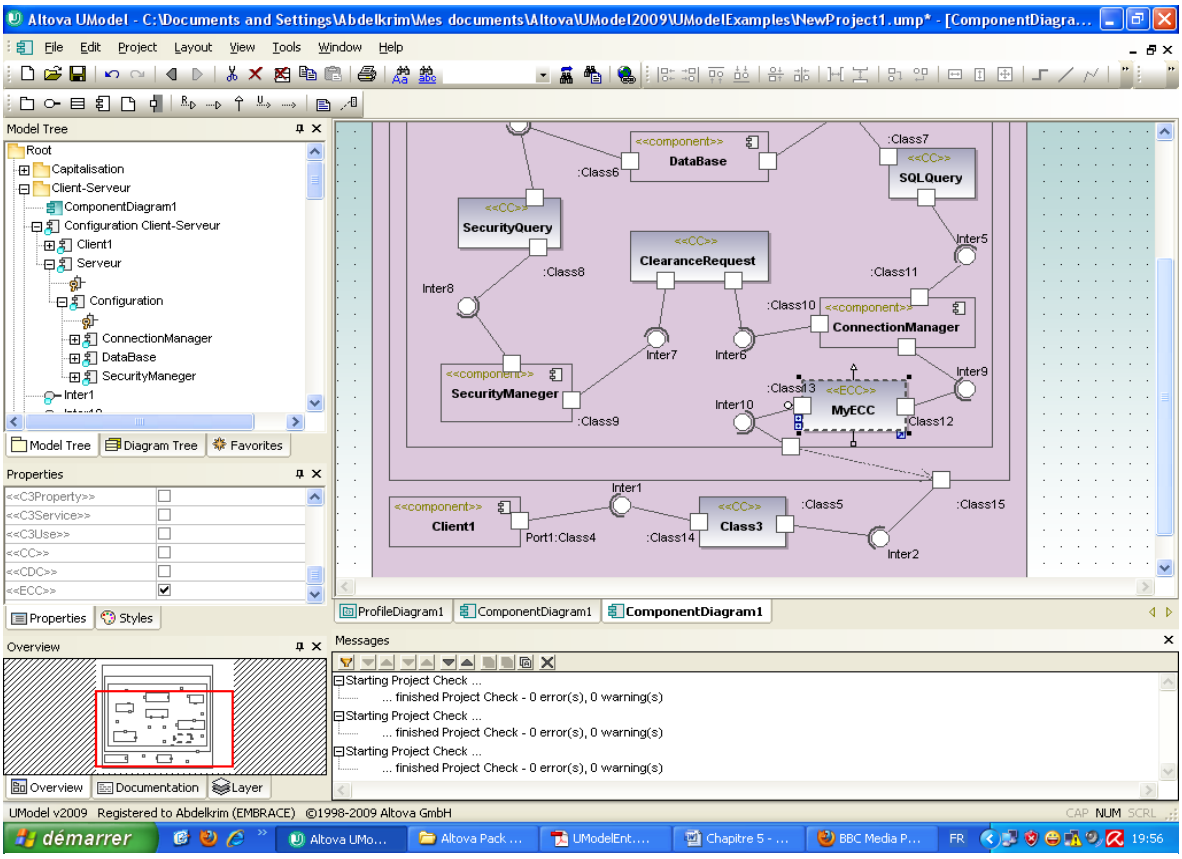
A.1.1 Extrait du code de l'application Client-Serveur en Java

```
package Configuration_CS;
import java.io.*;
import java.net.*;
import java.sql.*;

import Client_Serveur_Interface.*;

public class Client_Serveur implements ConfigServiceExterne{

public class Serveur implements IReceiverequest, ISendResponse, IExterne {
      public boolean password_needed = true;
      public String data_type = "formate-2";

      public class Connection_Manager implements
             IDBQueryInf,Isocket,IsecurityChek {
                  public Connection_Manager (){}
                  public void finalize () throws Throwable{}
      }

      public class SQL_Query implements Icaller, Icallee {

      public class Glu_SQL implements In_SQL, Out_SQL {
            public Glu_SQL () {}
            public void finalize () throws Throwable {}
      }
}

      public class DataBase implements IQueryInf, IsecurutyManagmentInf {
            public DataBase () {}
            public void finalize () throws Throwable {}
      }
```

```
public class Security_Query implements IRequestor, ICredentialQuery{

      public class Glu_Sec implements Out_Sec, In_Sec {
            public Glu_Sec (){}
            public void finalize () throws Throwable {}
      }
      public Security_Query (){}
            public void finalize () throws Throwable {}
      }

      public class Security_Manager implements ICerdentalQuery,
                  ISecurityAutorization {
            public Security_Manager (){}
            public void finalize () throws Throwable {}
      }

      public class ClearanceRequest implements IRequestor, IGrantor{

            public class Glu_Cle implements Out_Cle, In_Cle {
                  public Glu_Cle () {}
                  public void finalize () throws Throwable {}
            }
            public ClearanceRequest () {}
            public void finalize () throws Throwable {}
      }
      public Serveur() {}
      public void finalize () {}
}


public class RPC implements Icaller, Icallee {
      public String service_type = "communication";

      public class Glu implements Out_Glu_RPC, In_Glu_RPC {
            public Glu(){ }
            public void Role1 () {}
            public void Role2 () {}
            public void Communication (){
                  Role1();
                  Role2();
            }
            public void finalize(){};
      }
      public RPC() { }
      public void finalize(){}
}


public class Client implements ISendRequest, IReceiveResponse {
      public String data_type = "formate-2";
      public float request_rate = 21;
      public Client (){}
      public void finalize (){}
}
```

```
    public Client_Serveur (){}

    public void finalize () throws Throwable{}

    public static void main(String [] args ) {
          System.out.println ("Instantiation de la configuration client-
                               serveur");
          Client_Serveur arch1 = new Client_Serveur();
          Client_Serveur.Client C1;
          Client_Serveur.Serveur S1;
          Client_Serveur.RPC C1_S1;
          /* ------  définir ici les défférentes connexions  ----- */
    }
}
```

A.1.2 Conception de l'application Client-Serveur en UModel



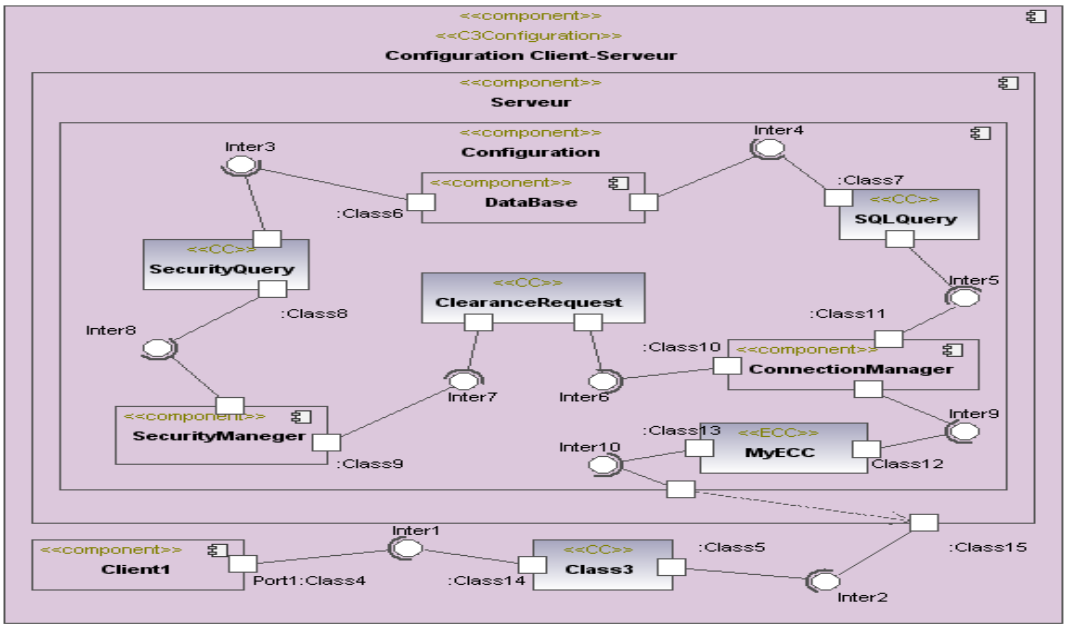


Figure A.1- Application Client-Serveur en UModel.

A.2 Application Pipe-Filter

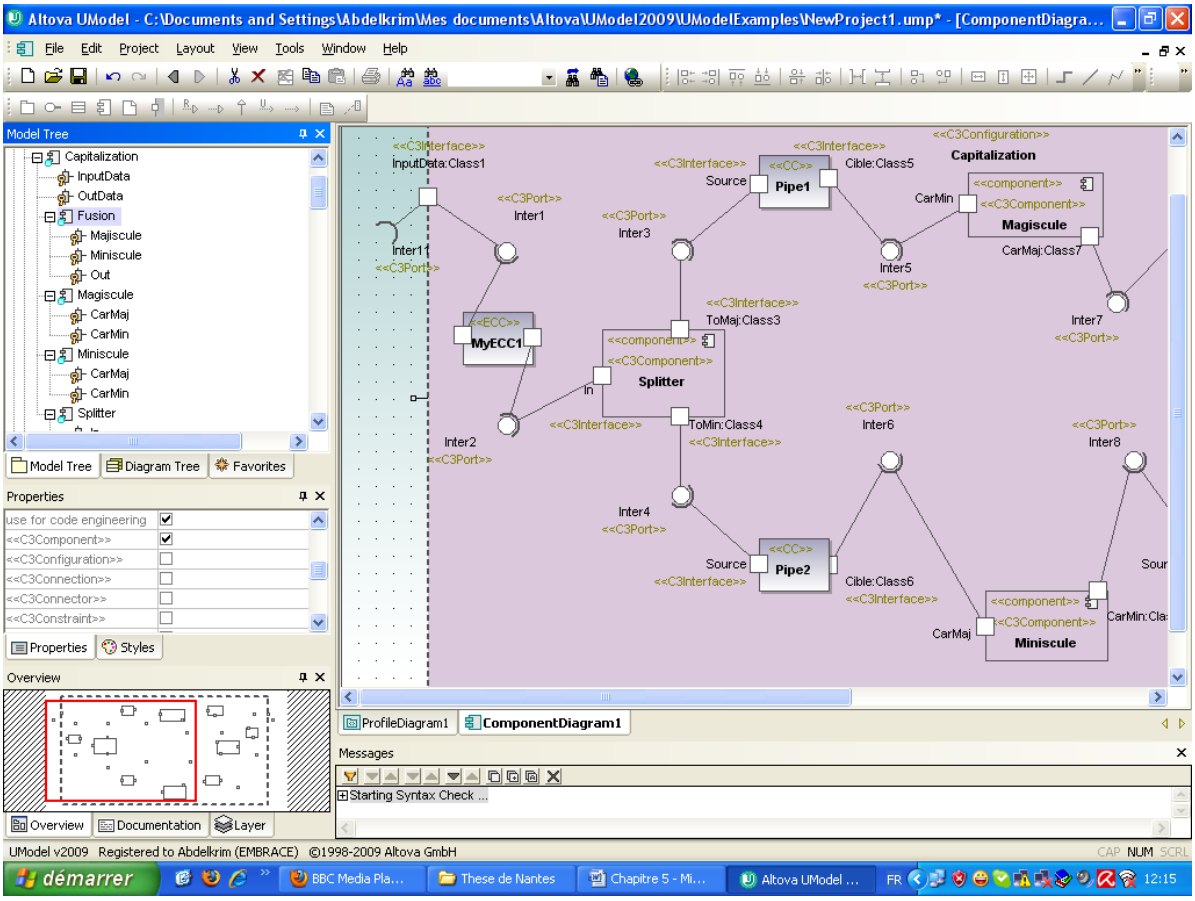
A.2.1 Extrait du code de l'application Pipe-Filter en Java

```
package Pipe-Filter;
public class Capitalize {
      static int buf;
      static String data;
      public Capitalize (String string1, int int1) {}
      public String capitalize () {return null;}
}

package Capitalisation;
public class Main {
      public Main(){}
      public static void main(String[] stringArray){}
}

package adl.composant;
public abstract class Composant {
      protected String name;
}

package adl.connecteur;
public abstract class Connecteur {
      protected String name;

}

package adl.composant;

import java.io.PipedReader;
import java.io.PipedWriter;

public class Filter extends Composant{

      public PipedReader portIn1;
      public PipedWriter portOut1;
      protected char[] cbuf;

      private void glue(){}
      public void cbuf_flush(){}
      public void print_cbuf(){}
      protected boolean isLastPacket(char [] charArray){return false;}
      protected void send (PipedWriter pipedWriter){}
      protected void receive(PipedReader pipedReader){}

}
```

```
package adl.connecteur;

import java.io.PipedReader;
import java.io.PipedWriter;

public class Pipe extends Connecteur{

      protected PipedReader compSource;
      protected PipedWriter compCible;
      protected char[] cbuf;

      public Pipe(String string, PipedWriter pipedWriter, PipedReader
      pipedReader, int int3){}

      public void role1_Recevoir (){}
      public void role2_Emettre (){}
      public void cbuf_flush(){}

      protected boolean isLastPacket(char [] charArray){return false;}
      public void print_cbuf(){}
      public void run() {}

}

package adl.composant;
import java.io.PipedWriter;

public class Source extends Composant{
      private char[] cbuf;
      private int buf;
      public PipedWriter portOut1;
      public Source(String string, String string1, int int2){}
      public void run(){}

}

package adl.composant;
import java.io.PipedWriter;

public class Splitter extends Filter{

      public PipedWriter pourOut2;
      public Splitter(String string, int int1, Source source){}
      public void run() {}

}

package adl.composant;
public class Maj extends Filter{
      public Maj (String string, int int1){}
      public void run(){}

}
```

```
package adl.composant;

public class Min extends Filter {

	public Min(String string, int int1){}
	public void run(){}

}

package adl.composant;
import java.io.PipedReader;

public class Target extends Composant{

	private String texte;
	private char[] cbuf;
	public PipedReader portIn1;
	public Target(String string, int int1, Fusion fusion){}
	public void cbuf_flush(){}
	protected boolean isLastPacket(char [] charArray){return false;}
	public String returnData() {return null;}
	public void run(){}

}

package adl.composant;
import java.io.PipedReader;

public class Fusion extends Filter{

	public PipedReader portIn2;
	public Fusion(String string, int int1){}
	public void run(){}

}
```

A.2.2 Conception de l'application Pipe-Filter en UModel



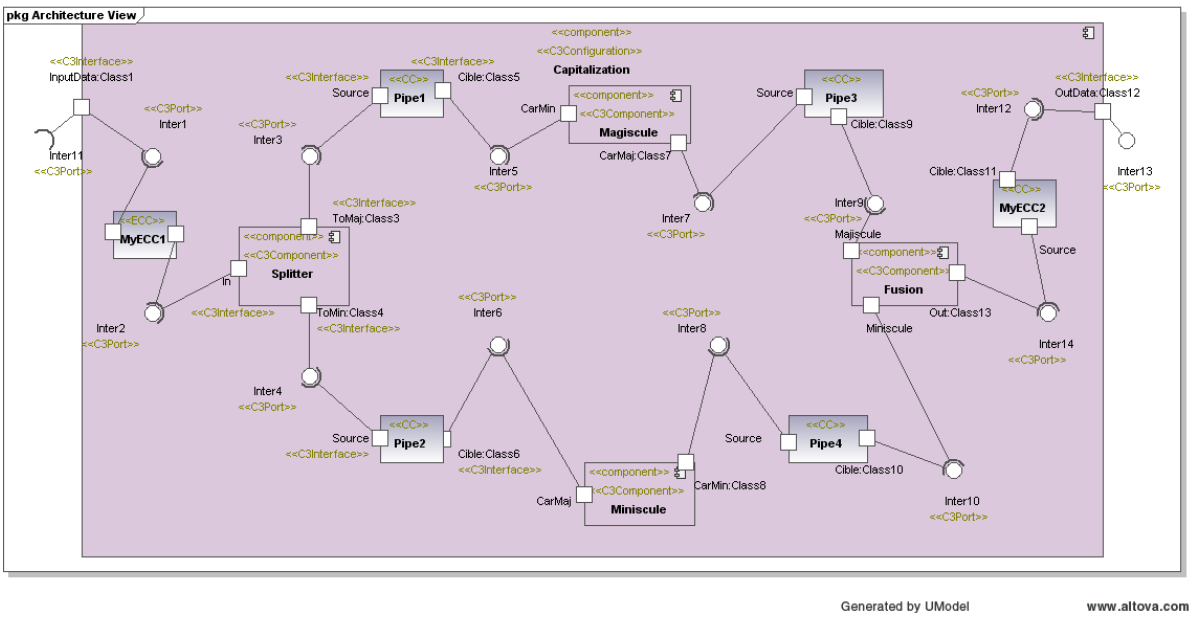


Figure A.2- Application Capitalisation en UModel.

A.3 Diagramme de profil de C3-UML en UModel/Altova

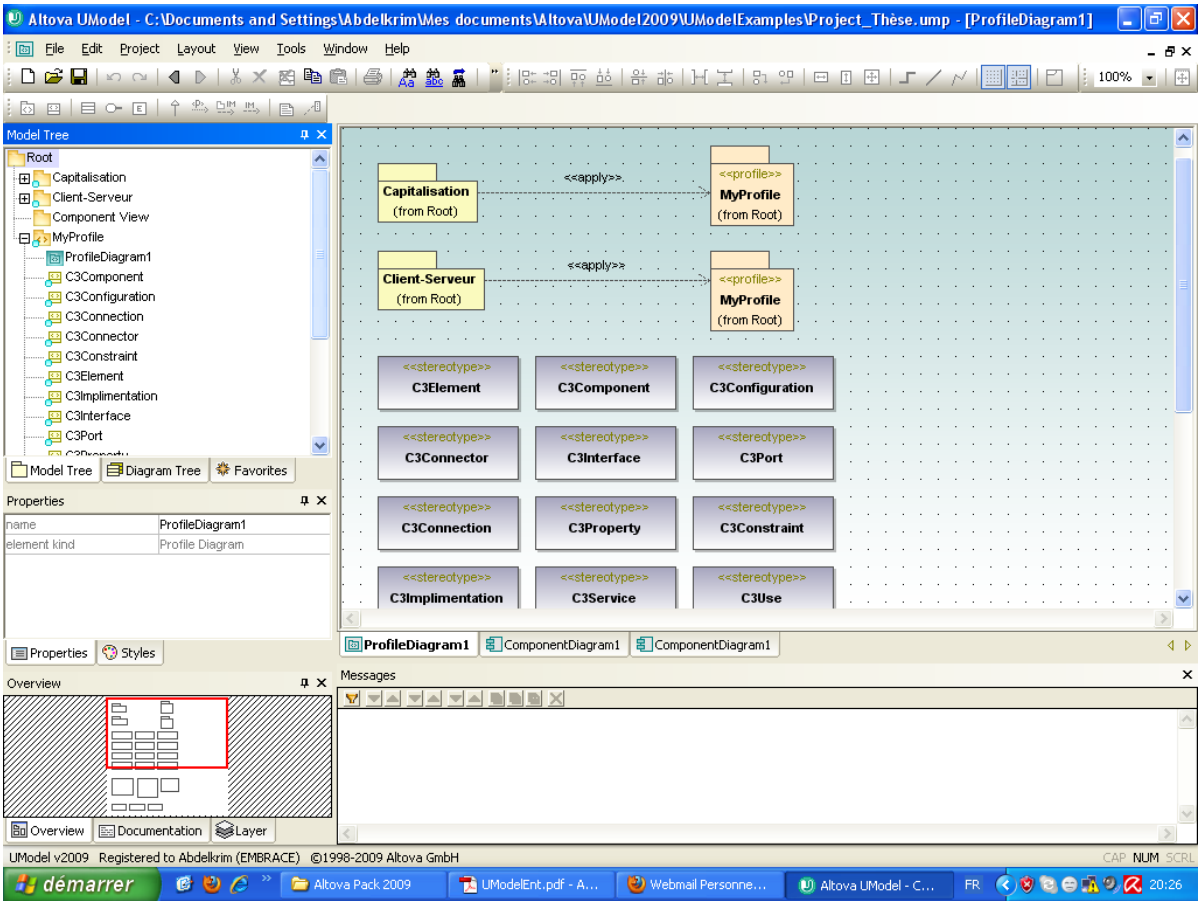


Figure A.3- C3-UML en UModel/Altova.

ANNEXE B

MADL : Un méta langage de description d'architecture

B.1 Introduction

MADL (pour *Meta Architecture Description Language*) est un méta langage de description d'architectures logicielles, qui abstrait les concepts des architectures (composants, connecteurs, configurations). Il facilite la manipulation, la réutilisation et l'évolution des architectures logicielles, comme il permet la transformation et la comparaison entre les ADLs [Smeda *et al.* 2008].

B.2 Méta-modélisation

Un acte de méta-modélisation a les mêmes objectifs qu'un acte de modélisation avec pour seule différence l'objet de la modélisation. Dans le cas d'un modèle réflexif, la méta-modélisation permet à un modèle de s'auto décrire : il est à la fois l'enjeu et le moyen de modélisation.

Dans le contexte des architectures logicielles, la méta-modélisation est un acte de modélisation appliqué sur une architecture. Le résultat d'un acte de modélisation d'un système S (c'est-à-dire, établir son abstraction) est une architecture. De la même manière, la méta architecture d'une architecture est une architecture qui modélise non pas un système réel mais une architecture. Une méta-architecture est donc un ADL. La méta-architecture étant elle-même une architecture et donc elle peut être modélisée. On obtient alors l'architecture de la méta-architecture : "*la méta-méta-architecture*". Comme dans toute modélisation récurrente, il convient de s'arrêter sur une architecture réflexive, c'est-à-dire qui s'auto décrit. Le nombre de niveaux importe peu, mais il semble que trois niveaux de modélisation soient amplement suffisants dans un cadre d'ingénierie d'architectures où la méta-méta-architecture servira à s'auto-modéliser et à modéliser les ADLs [Smeda 2006].

B.3 MADL (Meta Architecture Description Language)

Les quatre niveaux de méta-modélisation de l'OMG [OMG 2002a] peuvent être appliqués à l'architecture logicielle. Le résultat est une structure à quatre niveaux conceptuels ; le niveau application réelle A0, le niveau architecture A1, le niveau méta-architectureA2 et le niveau méta-méta-architecture A3, comme le montre le Tableau B.1.

	Modélisation par objet	Modélisation par composant
Niveau méta-méta-modèle (M3)	MOF	MADL
Niveau méta-modèle (M2)	UML	C3, Fractal, C2, ACME, etc.
Niveau modèle (M1)	Modèles	Architectures
Niveau instance (M0)	Instances	Instances

Tableau B.1 – Les niveaux conceptuels dans la modélisation par objet et par composant.

A l'instar du MOF, la méta-méta-architecture joue un rôle de solution unifiée de représentation d'architectures. Elle permet de structurer ces architectures, de faciliter la transformation d'architectures et de favoriser le passage de l'une vers l'autre. Les concepts de base de la méta-méta-architecture sont : méta-composant, méta-connecteur, méta-configuration et méta-interface. Ces méta-concepts sont organisés dans une méta-méta-architecture appelée MADL [Smeda *et al.* 2008], comme le montre la Figure B.1. MADL est méta-modélisé en lui-même « *réflexif* ». Cette disposition est destinée à mettre fin à l'empilement de modèles d'architectures.

Le modèle MADL est organisé en trois paquetages : méta-méta-architecture(M^2A), méta-architecture(MA), et architecture (A). Le paquetage M^2A traduit le fait que toute architecture doit dériver d'une MA. Le paquetage MA classifie et définit des architectures et il contient les méta-éléments d'architectures (méta-composant, méta-connecteurs et méta-interface). Le paquetage architecture hérite de composant pour respecter le principe « tout est un composant » dans MADL.

Principes de base :

1. MADL est orienté composant dans le sens où tout est composant (tous les éléments sont des sous-types de la classe abstraite *Composant*).

2. Chaque architecture doit être explicitement dérivée d'une méta-architecture.

3. Chaque architecture est une instance de son architecture supérieure, à l'exception de la méta-méta-architecture, qui est une instance d'elle-même.

4. Les dépendances entre les architectures et les éléments sont basées sur les dépendances utilisées dans le modèle MOF, à savoir : l'instanciation, l'héritage et la composition.

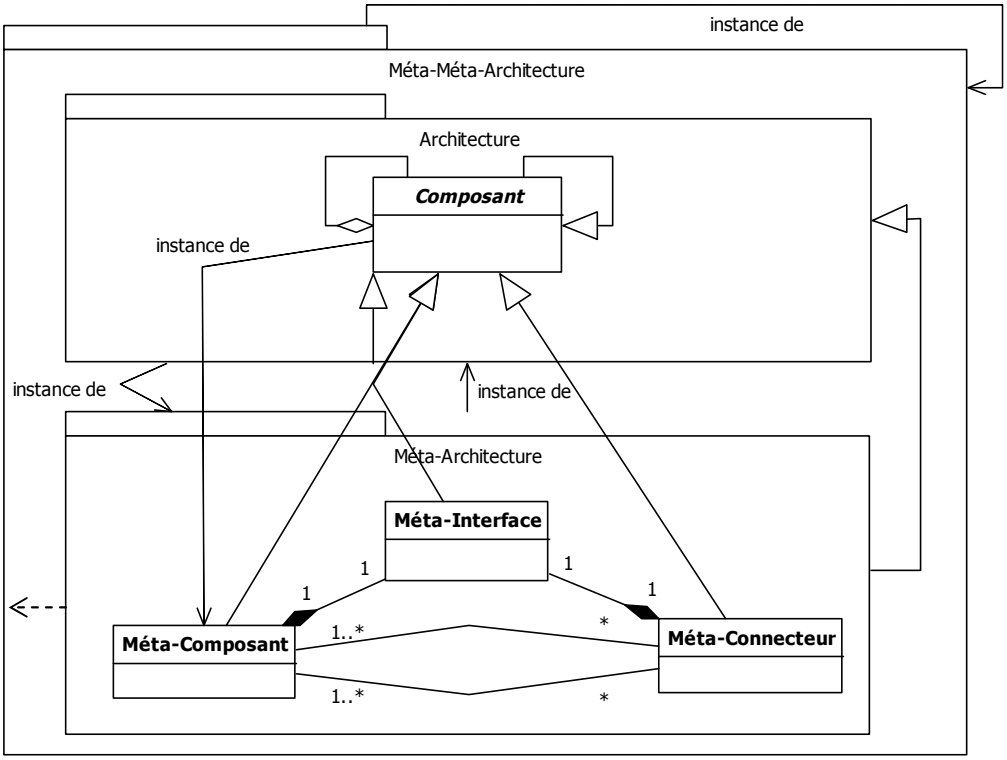


Figure B.1- La structure du modèle MADL.

B.4 Instanciation de MADL : l'exemple C3

La définition d'une méta-architecture peut s'appréhender de deux manières différentes :

- l'*instanciation*, opération qui, à partir d'une méta-architecture, permet de créer des architectures,
- la *représentation* d'une architecture, opération qui, à partir d'une architecture, permet de créer une méta-architecture.

Pour définir une nouvelle méta-architecture (un nouveau ADL), MADL est instancié, et un nouveau modèle conforme à la définition de MADL est obtenu. Chaque élément de la méta-architecture est une instance d'un élément de MADL. La Figure B.2 montre comment MADL peut être instancié pour obtenir le méta-modèle C3. Comme le montre cette figure, chaque élément de C3 doit être conforme à un élément de MADL. Les composants et les configurations sont des instances de *Méta-Composant*, les connecteurs sont instanciés à partir de *Méta-Connecteur* et les interfaces de composants et de connecteurs sont instanciées à partir de *Méta-Interface*. En conclusion C3 est une instance de MADL.

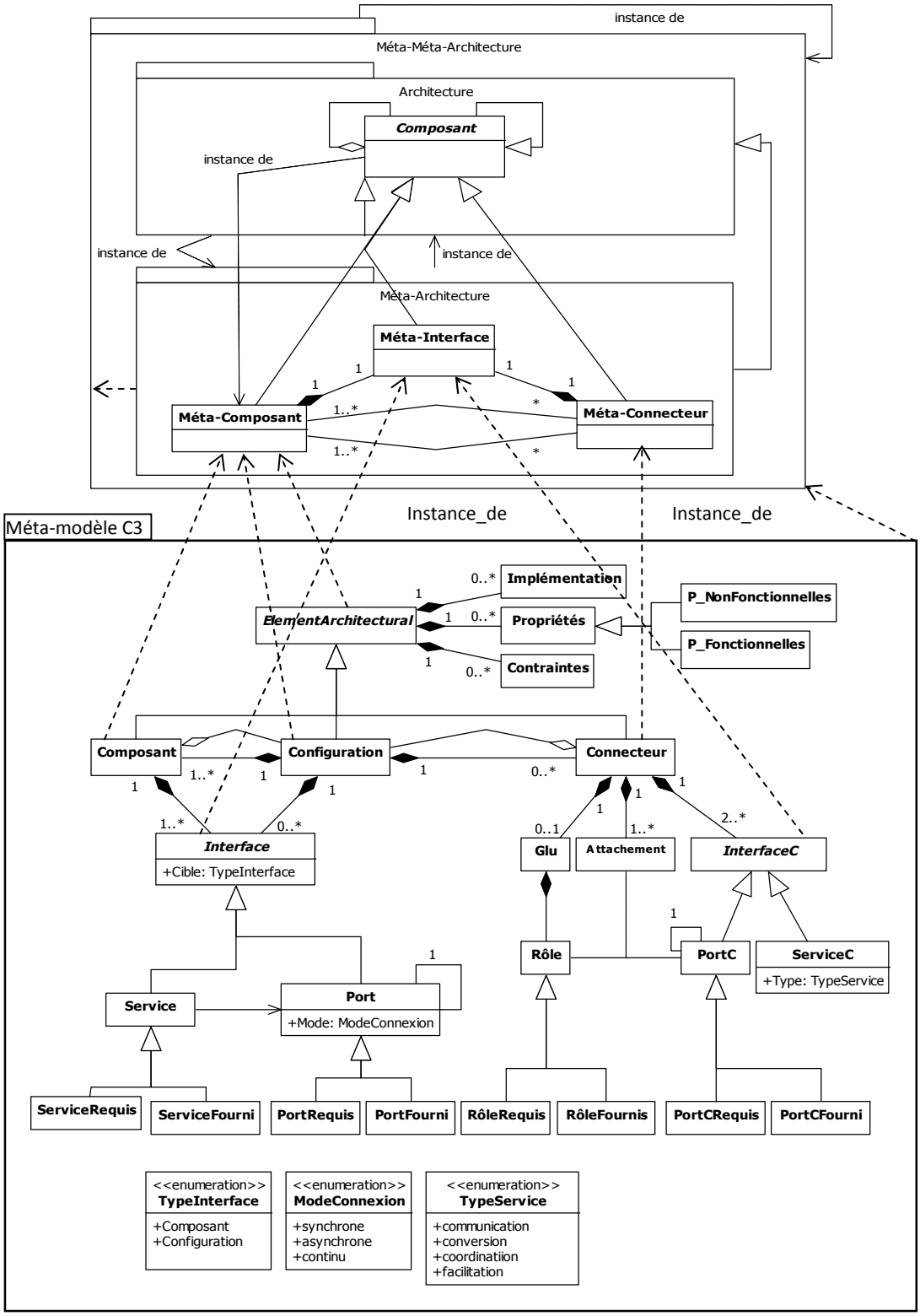


Figure B.2- L'instanciation de MADL pour obtenir C3.

La Figure B.3 montre la projection entre les concepts de MADL et ceux du MOF. Même avec l'introduction de nouvelles relations, les critères mentionnés ci-dessus sont pris en compte et respectés.

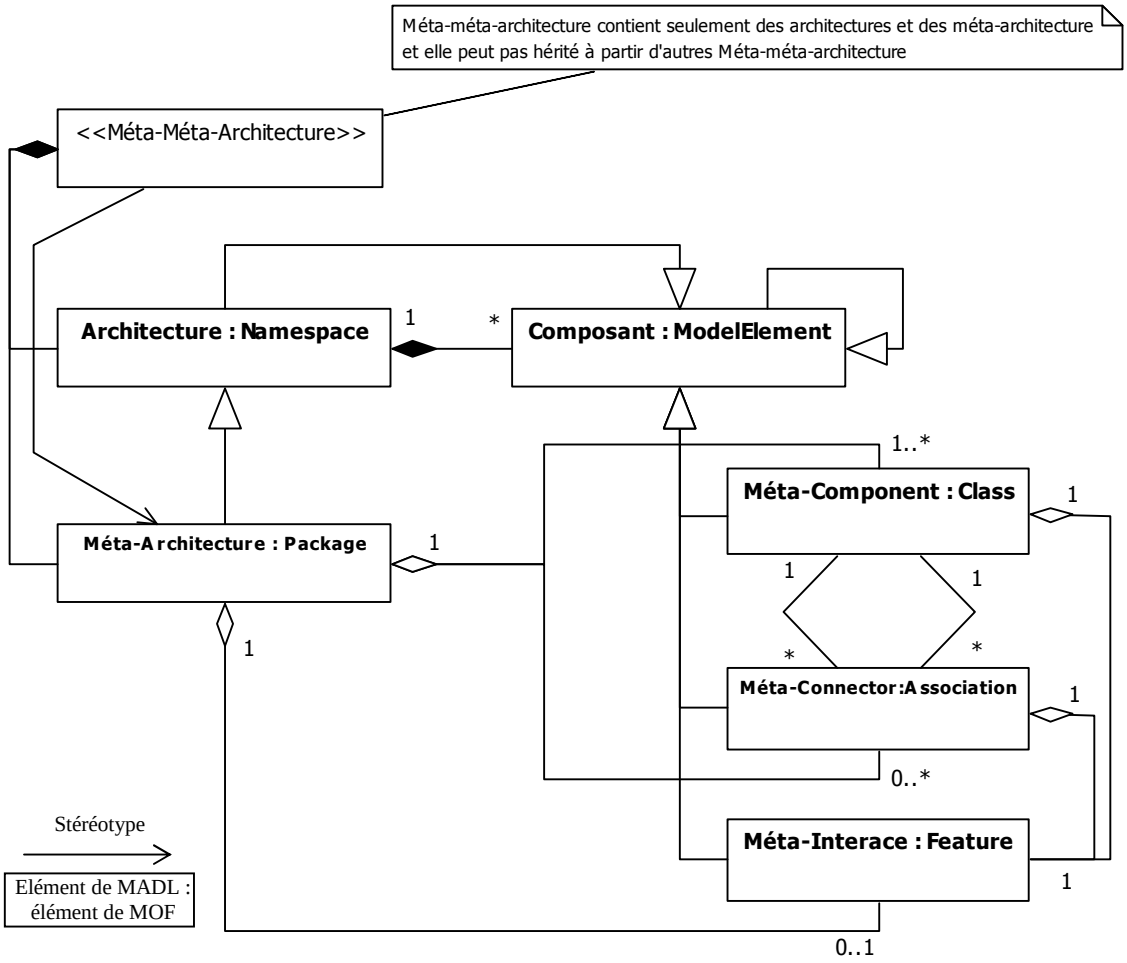


Figure B.3- Projection de MADL vers MOF.

Contribution à l'élaboration d'architectures logicielles à hiérarchies multiples

Abdelkrim AMIRAT

Résumé

En favorisant leurs descriptions à un haut niveau d'abstraction, les architectures logicielles ont été introduites en réponse à l'accroissement de la complexité des systèmes. L'architecture d'un système logiciel fournit un modèle du système qui masque les détails d'exécution, permettant à l'architecte de se concentrer sur l'analyse et les décisions les plus cruciales afin de répondre au mieux aux exigences dudit système. Dans cette thèse, nous proposons d'aborder la problématique de description d'architectures logicielles à hiérarchies multiples, avec comme objectif, d'offrir aux concepteurs plusieurs vues architecturales sur le système en exploitant le mieux possible la granularité et la spécificité des composants, des connecteurs et des configurations, et en favorisant leur réutilisation.

Notre contribution se résume en trois volets majeurs. Le premier concerne la proposition d'un ADL (Architecture Description Language) baptisé C3 reposant d'une part sur un modèle de représentation s'appuyant sur les concepts explicites de composants, de connecteurs et de configurations et d'autre part sur un modèle de raisonnement basé sur quatre types de hiérarchies (*structurelle*, *fonctionnelle*, *conceptuelle* et *de méta-modélisation*) pour décrire les architectures logicielles à différents niveaux de compréhension. Le deuxième volet concerne la proposition du modèle MY comme une méthodologie à suivre pour décrire les architectures logicielles à base de composants. Cette démarche décrit les concepts architecturaux selon un triptyque : composant, connecteur et configuration. Enfin, Le troisième volet concerne le développement d'un profil UML (C3-Profil) qui permet de faire la projection des architectures définies en C3 vers UML 2.0 afin de profiter des outils supports d'UML 2.0.

Mots-clés : Architecture logicielle, Langages de description d'architectures, Profil UML, Connecteur, Configuration, Gestionnaire de connexions, Hiérarchies multiples.

Abstract

By promoting their descriptions at a high level of abstraction, software architectures have been introduced in response to increasing complexity of systems. The architecture of a software system provides a model system that hides the details of execution, allowing the architect to focus on the analysis and the most crucial decisions to best meet the requirements of that system. In this thesis, we propose to tackle the problem of multiple hierarchies' description of software architectures, with the aim to offer designers several architectural views on the system by exploiting the granularity and the specificity of components, connectors and configurations, and encouraging their reuse.

Our contribution is summarized in three major aspects. The first concerns the proposal for an ADL (Architecture Description Language) known as C3 based in part on a representation model supported by explicit concepts of components, connectors and configurations, and the other on a reasoning model supported by four types of hierarchies (structural, functional, conceptual, and meta-modeling) to describe software architectures at different levels of understanding. The second aspect concerns the proposal of MY model as a methodology to describe the core concepts of software architectures (component, connector, and configuration). Finally, the third aspect involves the development of an UML profile (C3-Profile) which allows the projection of C3 architectures to UML 2.0 description to take advantage of tools support associated with UML 2.0.

Keywords: Software architecture, Architecture description languages, UML Profile, Connector, Configuration, Connection Manager, Multi hierarchies.